Advanced IceSSL Topics

This page discusses some additional capabilities of the IceSSL plug-in.

On this page:

Managing Certificate Passwords

IceSSL may need to obtain a password if it loads a file that contains secure data, such as an encrypted private key. An application can supply a plain-text password in a configuration property, but doing so is a potential security risk. For example, if you define the property on the application's command-line, it may be possible for other users on the same host to see the password simply by obtaining a list of active processes. If you define the property in a configuration file, the password is only as secure as the file in which it is defined.

In highly secure environments where access to a host is tightly restricted, a password can safely be supplied as a plain-text configuration property, or the need for the password can be eliminated altogether by using unsecured key files.

In situations where password security is a concern, the application generally needs to take additional action.

Dynamic Properties

A common technique is to prompt the user for a password and transfer the user's input to a configuration property that the application defines dynamically, as shown below:

C++

```
string password = // ...
Ice::InitializationData initData;
initData.properties = Ice::createProperties(argc, argv);
initData.properties->setProperty("IceSSL.Password", password);
Ice::CommunicatorPtr comm = Ice::initialize(initData);
```

The password must be present in the property set before the communicator is initialized, since IceSSL needs the password during its initialization, and the communicator initializes plug-ins automatically by default.

Password Callbacks in C++

If a password is required but the application has not configured one, IceSSL prompts the user at the terminal during the plug-in's initialization. This behavior is not suitable for some types of applications, such as a program that runs automatically at system startup as a Unix daemon or Windows service.

A terminal prompt is equally undesirable for graphical applications, which would generally prefer to prompt the user in an application window. The dynamic property technique described in the previous section is usually appropriate in this situation.

If your application must supply a password, and you do not want to use a configuration property or a terminal prompt, your remaining option is to install a `PasswordPrompt` object in the plug-in using the `setPasswordPrompt` method. The `PasswordPrompt` class is defined as follows:

C++

```
namespace IceSSL
{
class PasswordPrompt : public IceUtil::Shared
{
public:

    virtual std::string getPassword() = 0;
};
typedef IceUtil::Handle<PasswordPrompt> PasswordPromptPtr;
}
```

IceSSL invokes `getPassword` on the object when a password is required. If the object returns an incorrect password, IceSSL tries again, up to the limit defined by the `IceSSL.PasswordRetryMax` property.

Note that you must delay the initialization of the IceSSL plug-in until after the `PasswordPrompt` object is installed. To illustrate this point, consider the following example:

C++

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr = communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin = IceSSL::PluginPtr::dynamicCast(plugin);
sslPlugin->setPasswordPrompt(new Prompt); // OOPS!
```

This code is incorrect because the `PasswordPrompt` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to load the file that required a password.

The correct approach is to define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

C++

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr = communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin = IceSSL::PluginPtr::dynamicCast(plugin);
sslPlugin->setPasswordPrompt(new Prompt);
pluginMgr->initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordPrompt` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Password Callbacks in Java

If you do not want to use configuration properties to define passwords, you can install a `PasswordCallback` object in the plug-in using a configuration property, or using the `setPasswordCallback` method. The `PasswordCallback` interface has the following definition:

Java

```
public interface PasswordCallback
{
    char[] getPassword(String alias);
    char[] getTruststorePassword();
    char[] getKeystorePassword();
}
```

The methods are described below:

- `getPassword`
  Supplies the password for the key with the given alias. The return value must not be null.
- `getTruststorePassword`
  Supplies the password for a truststore. The method may return null, in which case the integrity of the truststore is not verified.
- `getKeystorePassword` to obtain the password for a keystore.
  Supplies the password for a keystore. The method may return null, in which case the integrity of the keystore is not verified.

For each of these methods, IceSSL clears the contents of the returned array as soon as possible.

The simplest way to install the callback is by defining the configuration property `IceSSL.PasswordCallback`. The property's value is the name of your callback implementation class. IceSSL instantiates the class using its default constructor.

To install the callback manually, you must delay the initialization of the IceSSL plug-in until after the `PasswordCallback` object is installed. To illustrate this point, consider the following example:

Java

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI()); // OOPS!
```

This code is incorrect because the `PasswordCallback` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to retrieve the certificate that required a password.

The correct approach is to define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

Java

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI());
pluginMgr.initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordCallback` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Password Callbacks in .NET

If you do not want to use configuration properties to define passwords, you can install a `PasswordCallback` object in the plug-in using a configuration property, or using the `setPasswordCallback` method. The `PasswordCallback` interface has the following definition:

C#

```
using System.Security;

public interface PasswordCallback
{
    SecureString getPassword(string file);
    SecureString getImportPassword(string file);
}
```

The methods are described below:

- `getPassword`
  Supplies the password for the given file. The method may return null if no password is required.
- `getImportPassword`
  Supplies the password for a file from which certificates are imported into the certificate store. The method may return null if no password is required.

The simplest way to install the callback is by defining the configuration property `IceSSL.PasswordCallback`. The property's value is the name of your callback implementation class. IceSSL instantiates the class using its default constructor.

To install the callback manually, you must delay the initialization of the IceSSL plug-in until after the `PasswordCallback` object is installed. To illustrate this point, consider the following example:

C#

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI()); // OOPS!
```

This code is incorrect because the `PasswordCallback` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to retrieve the certificate that required a password.

The correct approach is to define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

C#

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI());
pluginMgr.initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordCallback` object, the application invokes `in itializePlugins` on the plug-in manager to complete the plug-in initialization process.

Manually Configuring IceSSL

The `Plugin` interface supports a method in each of the supported language mappings that provides an application with more control over the plug-in's configuration.

In C++ and Java, an application can call the `setContext` method to supply a pre-configured "context" object used by the underlying SSL engines. In .NET, the `setCertificates` method accepts a collection of certificates that the plug-in should use. In all cases, using one of these methods causes IceSSL to ignore (at a minimum) the configuration properties related to certificates and keys. The application is responsible for accumulating its certificates and keys, and must also deal with any password requirements.

Describing the use of these plug-in methods in detail is outside the scope of this manual, however it is important to understand their prerequisites. In particular, the application needs to have the communicator load the plug-in but not actually initialize it until after the application has had a chance to interact directly with it. (The previous section showed examples of this technique.) The application must define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

With this setting, the application becomes responsible for completing the plug-in initialization process by invoking `initializePlugins` on the `PluginManager`. The C# example below demonstrates the proper steps:

C#

```
Ice.Communicator comm = // ...
Ice.PluginManager pluginMgr = comm.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
X509Certificate2Collection certs = // ...
sslPlugin.setCertificates(certs);
pluginMgr.initializePlugins();
```

Using Custom Plug-ins with IceSSL

The Ice plug-in facility is not restricted to protocol implementations. Ice only requires that a plug-in implement the `Ice::Plugin` interface and support the language-specific mechanism for dynamic loading.

The customization options of the IceSSL plug-in make it possible for you to install an application-specific implementation of a certificate verifier in an existing program. For example, you could install a custom certificate verifier in a Glacier2 router without the need to modify Glacier2's source code or rebuild the executable. You would have to write a C++ plug-in to accomplish this, since Glacier2 is written in C++. In short, your plug-in must interact with the IceSSL plug-in and install a certificate verifier.

For this technique to work, it is important that the plug-ins be loaded in a particular order. Specifically, the IceSSL plug-in must be loaded first, followed by the certificate verifier plug-in. By default, Ice loads plug-ins in an undefined order, but you can use the property `Ice.PluginLoadOrder` to specify a particular order.

As an example, let's write a plug-in that installs our simple C++ certificate verifier. Here is the definition of our plug-in class:

C++

```
class VerifierPlugin : public Ice::Plugin
{
public:
    VerifierPlugin(const Ice::CommunicatorPtr & comm) :
        _comm(comm)
    {
    }

    virtual void initialize()
    {
        Ice::PluginManagerPtr pluginMgr = _comm->getPluginManager();
        Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
        IceSSL::PluginPtr sslPlugin = IceSSL::PluginPtr::dynamicCast(plugin);
        sslPlugin->setCertificateVerifier(new Verifier);
    }

    virtual void destroy()
    {
    }

private:
    Ice::CommunicatorPtr _comm;
};
```

The class implements the two operations in the `Plugin` interface, `initialize` and `destroy`. The code in `initialize` installs the certificate verifier object, while nothing needs to be done in `destroy`.

The next step is to write the plug-in's factory function, which the communicator invokes to obtain an instance of the plug-in:

C++

```
extern "C"
{

Ice::Plugin*
createVerifierPlugin(
    const Ice::CommunicatorPtr & communicator,
    const string & name,
    const Ice::StringSeq & args)
{
    return new VerifierPlugin(communicator);
}

}
```

We can give the function any name; in this example, we chose `createVerifierPlugin`.

Finally, to install the plug-in we need to define the following properties:

C++

```
Ice.PluginLoadOrder=IceSSL,Verifier
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.Verifier=Verifier:createVerifierPlugin
```

The value of `Ice.PluginLoadOrder` guarantees that IceSSL is loaded first. The plug-in specification `Verifier:createVerifierPlugin` identifies the name of the shared library or DLL and the name of the registration function.

There are a few more details you must attend to, such as ensuring that the factory function is exported properly and building the shared library or DLL that contains the new plug-in. Our discussion of the plug-in facility provides more information on developing a plug-in.

See Also

- The Server-Side main Function in C++
- Configuring IceSSL
- Programming IceSSL in C++
- Programming IceSSL in Java
- Programming IceSSL in .NET
- Plug-in Facility
- IceSSL Properties
- Ice Plug-In Properties