Programming IceSSL in C++

This page describes the C++ API for the IceSSL plug-in.

On this page:

The IceSSL Plugin Interface in C++

Applications can interact directly with the IceSSL plug-in using the native C++ class `IceSSL::Plugin`. A reference to a `Plugin` object must be obtained from the communicator in which the plug-in is installed:

C++

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

The `Plugin` class supports the following methods:

C++

```
namespace IceSSL
{
class Plugin : public Ice::Plugin
{
public:
    virtual void setContext(SSL_CTX*) = 0;
    virtual SSL_CTX* getContext() = 0;

    virtual void setCertificateVerifier(const CertificateVerifierPtr&) = 0;

    virtual void setPasswordPrompt(const PasswordPromptPtr&) = 0;
};
typedef IceUtil::Handle<Plugin> PluginPtr;
}
```

The `setContext` and `getContext` methods are rarely used in practice. The `setCertificateVerifier` method installs a custom certificate verifier object that the plug-in invokes for each new connection. The `setPasswordPrompt` method provides an alternate way to supply IceSSL with passwords. We discuss certificate verifiers below and revisit the other methods in our discussion of advanced IceSSL programming.

Obtaining SSL Connection Information in C++

You can obtain information about any SSL connection using the `getInfo` operation on a `Connection` object. It returns an `IceSSL::NativeConnectionInfo` class instance that derives from the Slice class `IceSSL::ConnectionInfo`. The Slice base class is defined as follows:

Slice

```
module Ice {
    local class ConnectionInfo {
        bool incoming;
        string adapterName;
    };

    local class IPConnectionInfo extends ConnectionInfo {
        string localAddress;
        int localPort;
        string remoteAddress;
        int remotePort;
    };
};

module IceSSL {
    local class ConnectionInfo extends Ice::IPConnectionInfo {
        string cipher;
        Ice::StringSeq certs;
    };
};
```

In turn, the C++ class `NativeConnectionInfo` is defined as follows:

C++

```
class NativeConnectionInfo : public ConnectionInfo {
public:
    std::vector<CertificatePtr> nativeCerts;
};

typedef IceUtil::Handle<NativeConnectionInfo> NativeConnectionInfoPtr;
```

Installing a Certificate Verifier in C++

A new connection undergoes a series of verification steps before an application is allowed to use it. The low-level SSL engine executes certificate validation procedures and, assuming the certificate chain is successfully validated, IceSSL performs additional verification as directed by its configuration properties. Finally, if a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to allow the connection to proceed.

The `CertificateVerifier` interface has only one method:

C++

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

    virtual bool verify(const NativeConnectionInfoPtr&) = 0;
};
typedef IceUtil::Handle<CertificateVerifier> CertificateVerifierPtr;
}
```

IceSSL rejects the connection if `verify` returns `false`, and allows it to proceed if the method returns `true`. The `verify` method receives a `NativeConnectionInfo` object that describes the connection's attributes.

The `nativeCerts` member is a vector of certificates representing the peer's certificate chain. The vector is structured so that the first element is the peer's certificate, followed by its signing certificates in the order they appear in the chain, with the root CA certificate as the last element. The vector is empty if the peer did not present a certificate chain.

The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The local and remote address information is provided in `localAddress` and `remoteAddress`, respectively.

A bug in Windows XP prevents IceSSL from obtaining the remote address information when using IPv6.

The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

The following class is a simple implementation of a certificate verifier:

C++

```
class Verifier : public IceSSL::CertificateVerifier
{
public:

    bool verify(const IceSSL::NativeConnectionInfo& info)
    {
        if (!info.nativeCerts.empty())
        {
            string dn = info.nativeCerts[0].getIssuerDN();
            transform(dn.begin(), dn.end(), dn.begin(), ::tolower);
            if (dn.find("zeroc") != string::npos)
            {
                return true;
            }
        }
        return false;
    }
}
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

C++

```
IceSSL::PluginPtr sslPlugin = // ...
sslPlugin->setCertificateVerifier(new Verifier);
```

You should install the verifier before any SSL connections are established.

You can also install a certificate verifier using a custom plug-in to avoid making changes to the code of an existing application.

The Ice run time calls the `verify` method during the connection-establishment process, therefore delays in the `verify` implementation have a direct impact on the performance of the application. Do not make remote invocations from your implementation of `verify`.

Using Certificates in C++

The `ConnectionInfo` class contains a vector of `Certificate` objects representing the peer's certificate chain. `Certificate` is a reference-counted convenience class that hides the complexity of the underlying OpenSSL API. Its methods are inspired by the Java class `X509Certificate`:

C++

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

    Certificate(X509*);

    static CertificatePtr load(const string&);
    static CertificatePtr decode(const string&);

    bool operator==(const Certificate&) const;
    bool operator!=(const Certificate&) const;

    PublicKeyPtr getPublicKey() const;

    bool verify(const PublicKeyPtr&) const;

    string encode() const;

    bool checkValidity() const;
    bool checkValidity(const IceUtil::Time&) const;

    IceUtil::Time getNotAfter() const;
    IceUtil::Time getNotBefore() const;

    string getSerialNumber() const;

    DistinguishedName getIssuerDN() const;
    vector<pair<int, string> > getIssuerAlternativeNames();

    DistinguishedName getSubjectDN() const;
    vector<pair<int, string> > getSubjectAlternativeNames();

    int getVersion() const;

    string toString() const;

    X509* getCert() const;
};
typedef IceUtil::Handle<Certificate> CertificatePtr;
}
```

The more commonly-used methods are described below; refer to the documentation in `IceSSL/Plugin.h` for information on the methods that are not covered.

The static method `load` creates a certificate from the contents of a PEM-encoded file. If an error occurs, the function raises `IceSSL::CertificateReadException`; the reason member provides a description of the problem.

Use `decode` to obtain a certificate from a PEM-encoded string representing a certificate. The caller must be prepared to catch `IceSSL::CertificateEncodingException` if `decode` fails; the reason member provides a description of the problem.

The `encode` method creates a PEM-encoded string that represents the certificate. The return value can later be passed to `decode` to recreate the certificate.

The `checkValidity` methods determine whether the certificate is valid. The overloading with no arguments returns true if the certificate is valid at the current time; the other overloading accepts an `IceUtil::Time` object and returns true if the certificate is valid at the given time.

The `getNotAfter` and `getNotBefore` methods return instances of `IceUtil::Time` that define the certificate's valid period.

The methods `getIssuerDN` and `getSubjectDN` supply the distinguished names of the certificate's issuer (i.e., the CA that signed the certificate) and subject (i.e., the person or entity to which the certificate was issued). The methods return instances of the class `IceSSL::DistinguishedName`, another convenience class that is described below.

Finally, the `toString` method returns a human-readable string describing the certificate.

Using Distinguished Names in C++

X.509 certificates use a distinguished name to identify a person or entity. The name is an ordered sequence of relative distinguished names that supply values for fields such as common name, organization, state, and country. Distinguished names are commonly displayed in stringified form according to the rules specified by RFC 2253, as shown in the following example:

```
C=US, ST=Florida, L=Palm Beach Gardens, O="ZeroC, Inc.", OU=Servers, CN=Quote Server
```

`DistinguishedName` is a convenience class provided by IceSSL to simplify the tasks of parsing, formatting and comparing distinguished names.

C++

```
namespace IceSSL
{
class DistinguishedName
{
public:

    DistinguishedName(const std::string&);
    DistinguishedName(const std::list<std::pair<std::string, std::string> >&);

    bool operator==(const DistinguishedName&) const;
    bool operator!=(const DistinguishedName&) const;
    bool operator<(const DistinguishedName&) const;

    bool match(const DistinguishedName&) const;

    operator std::string() const;
};
}
```

The first overloaded constructor accepts a string argument representing a distinguished name encoded using the rules set forth in RFC 2253. The new `DistinguishedName` instance preserves the order of the relative distinguished names in the string. The caller must be prepared to catch `IceSSL::ParseException` if an error occurs during parsing.

The second overloaded constructor requires a list of type-value pairs representing the relative distinguished names. The new `DistinguishedName` instance preserves the order of the relative distinguished names in the list.

The overloaded operator functions `operator==`, `operator!=`, and `operator<` perform an exact match of distinguished names in which the order of the relative distinguished names is important. For two distinguished names to be equal, they must have the same relative distinguished names in the same order.

The `match` function performs a partial comparison that does not consider the order of relative distinguished names. If `N1` and `N2` are instances of `DistinguishedName`, `N1.match(N2)` returns true if all of the relative distinguished names in `N2` are present in `N1`.

Finally, the string conversion operator encodes the distinguished name in the format described by RFC 2253.

See Also

- Using Connections
- Public Key Infrastructure
- Configuring IceSSL
- Advanced IceSSL Topics