Programming IceSSL in C++

Previous Next

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

On this page:

The IceSSL Plugin Interface in C++

The IceSSL C++ plug-in has several implementations listed in the table below. Its API consists of implementation-independent classes directly in the `IceSS` namespace plus implementation-dependent classes in nested namespaces such as `IceSSL::OpenSSL.`

Underlying SSL/TLS Implementation	IceSSL Plug-in Class	IceSSL Certificate Class	Platform Availability
OpenSSL	`IceSSL::Plugin` `IceSSL::OpenSSL::Plugin`	`IceSSL::Certificate` `IceSSL::OpenSSL::Certificate`	Linux (default), Windows
SecureTransport	`IceSSL::Plugin`	`IceSSL::Certificate` `IceSSL::SecureTransport::Certificate`	macOS (default)
SChannel	`IceSSL::Plugin`	`IceSSL::Certificate` `IceSSL::SChannel::Certificate`	Windows (default)
UWP	`IceSSL::Plugin`	`IceSSL::Certificate` `IceSSL::UWP::Certificate`	UWP (default)

Each platform has a default implementation, indicated with (default) in the table above. Static functions on the `IceSSL::Certificate` class are supplied by the default platform-dependent implementation.

Applications can interact directly with the IceSSL plug-in using the C++ class `IceSSL::Plugin`. You can retrieve this `Plugin` object as shown below:

C++11

```
auto communicator = ...
auto pluginMgr = communicator->getPluginManager();
auto plugin = plugin->getPlugin("IceSSL");
auto sslPlugin = std::dynamic_pointer_cast<IceSSL::Plugin>(plugin);
```

C++98

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

The `Plugin` class provides the following functions:

C++11

```
namespace IceSSL
{
    class Plugin : public Ice::Plugin
    {
    public:
        virtual void setCertificateVerifier(std::function<bool(const std::shared_ptr<IceSSL::ConnectionInfo>&)
>) = 0;
        virtual void setPasswordPrompt(std::function<std::string()>) = 0;
        virtual std::shared_ptr<Certificate> load(const std::string&) const = 0;
        virtual std::shared_ptr<Certificate> decode(const std::string&) const = 0;
    };
}
```

```
// Only with OpenSSL implementation
//
namespace Ice
{
    namespace OpenSSL
    {
        class Plugin : public IceSSL::Plugin
        {
        public:
            virtual Ice::Long getOpenSSLVersion() const = 0;
            virtual void setContext(SSL_CTX*) = 0;
            virtual SSL_CTX* getContext() = 0;
        };
    }
}
```

C++98

```
namespace IceSSL
{
    class Plugin : public Ice::Plugin
    {
    public:
        virtual void setCertificateVerifier(const CertificateVerifierPtr&) = 0;
        virtual void setPasswordPrompt(const PasswordPromptPtr&) = 0;
        virtual CertificatePtr load(const std::string&) const = 0;
        virtual CertificatePtr decode(const std::string&) const = 0;
    };
}
```

```
// Only with OpenSSL implementation
//
namespace Ice
{
    namespace OpenSSL
    {
        class Plugin : public IceSSL::Plugin
        {
        public:
            virtual Ice::Long getOpenSSLVersion() const = 0;
            virtual void setContext(SSL_CTX*) = 0;
            virtual SSL_CTX* getContext() = 0;
        };
    }
}
```

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

The `load` function creates a `Certificate` object from a file in PEM encoded format. Likewise, the `decode` function creates a certificate from a string in the PEM encoding format. The most-derived type of the certificate object depends on the plug-in implementation you're using:

Plug-in	Certificate
OpenSSL	`IceSSL::OpenSSL::Certificate`
SChannel	`IceSSL::SChannel::Certificiate`
SecureTransport	`IceSSL::SecureTransport::Certificate`
UWP	`IceSSL::UWP::Certificate`

> `IceSSL::OpenSSL::Plugin` extends the base `IceSSL::Plugin` class and provides the `getOpenSSLVersion`, `setContext` and `getContext` functions that are specific to the OpenSSL implementation and are rarely used in practice. The `IceSSL.InitOpenSSL` property is typically used in conjunction with `setContext`.

Back to Top ^

Obtaining SSL Connection Information in C++

You can obtain information about any SSL connection using the `getInfo` operation on a `Connection` object. IceSSL defines the following type in Slice:

Slice

```
// from Ice/Connection.ice
module Ice
{
    local class ConnectionInfo
    {
        ConnectionInfo underlying;
        bool incoming;
        string adapterName;
        string connectionId;
    }
}

// from IceSSL/ConnectionInfo.ice
module IceSSL
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string cipher;
        ["cpp:type:std::vector<CertificatePtr>",
         "java:type:java.security.cert.Certificate[]",
         "cs:type:System.Security.Cryptography.X509Certificates.X509Certificate2[]"]
        Ice::StringSeq certs;
        bool verified;
    }
}
```

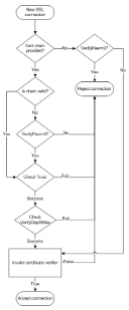
The `certs` member contains the peer's certificate chain as a sequence of `IceSSL::CertificatePtr` objects. The `cpp:type` metadata changes the default mapping of the `certs` member to that native type. The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The `verified` member indicates whether IceSSL was able to successfully verify the peer's certificate.

The inherited `underlying` data member contains the connection information of the underlying transport (if SSL is based on TCP, this member will contain an instance of `Ice::TCPEndpointInfo` which you can use to retrieve the remote and local addresses). The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). The `connectionId` data member matches the connection identifier set on the proxy. Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

Back to Top ^

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. If a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to accept or reject the connection. The value of the `IceSSL.VerifyPeer` property also plays an important role here. We've summarized the process in the following flow chart:



C++11

With the C++11 mapping, the certificate verifier is a `std::function` provided to the IceSSL plug-in's `setCertificateVerifier` member function:

```
virtual void setCertificateVerifier(std::function<bool(const std::shared_ptr<IceSSL::ConnectionInfo>&)>) = 0;
```

IceSSL rejects the connection if the certificate verifier function returns `false`, and allows it to proceed if the function returns `true`. The `verify` function receives a `IceSSL::ConnectionInfo` object that describes the connection's attributes.

The following code demonstrates a simple implementation of a certificate verifier:

```
auto verifier = [](const shared_ptr<IceSSL::ConnectionInfo>& info)
    {
        if(!info->certs.empty())
        {
            auto cert = IceSSL::Certificate::decode(info->certs[0]);
            auto dn = cert->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:

```
shared_ptr<IceSSL::Plugin> sslPlugin = ...
sslPlugin->setCertificateVerifier(verifier);
```

C++98

With the C++98 mapping, you must provide an object that implements the `CertificateVerifier` abstract base class:

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

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

IceSSL rejects the connection if the certificate verifier function returns `false`, and allows it to proceed if the function returns `true`. The `verify` function receives a `IceSSL::ConnectionInfo` object that describes the connection's attributes.

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

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

    bool verify(const IceSSL::ConnectionInfoPtr& info)
    {
        if(!info->certs.empty())
        {
            IceSSL::CertificatePtr cert = IceSSL::Certificate::decode(info->certs[0]);
            string dn = cert->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:

```
IceSSL::PluginPtr sslPlugin = ...
sslPlugin->setCertificateVerifier(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 certificate verifier during the connection-establishment process, therefore delays in the certificate verifier implementation have a direct impact on the performance of the application. Do not make remote invocations from your implementation of the certificate verifier.

Back to Top ^

Using Certificates in C++

The `IceSSL::ConnectionInfo` class contains a vector of `IceSSL::Certificate` objects representing the peer's certificate chain. `IceSSL::Certificate` is a reference-counted convenience class that hides the complexity of the platform's native SSL API inspired by the Java class `X509Certificate`. The `IceSSL::Certificate` class defines the common API for all IceSSL implementations, and APIs that use platform-dependent types are provided by extensions of this class. When a common method is not supported by a SSL implementation it raises `Ice::FeatureNotSupportedException`.

C++11

```
namespace IceSSL
{
    class Certificate : public std::enable_shared_from_this<Certificate>
    {
    public:

        virtual bool verify(const std::shared_ptr<Certificate>&) const = 0;
        virtual std::string encode() const = 0;

        virtual bool checkValidity() const = 0;
        virtual bool checkValidity(const std::chrono::system_clock::time_point&) const = 0;
        virtual std::chrono::system_clock::time_point getNotAfter() const = 0;
        virtual std::chrono::system_clock::time_point getNotBefore() const = 0;

        virtual std::string getSerialNumber() const = 0;

        virtual DistinguishedName getIssuerDN() const = 0;
        virtual std::vector<std::pair<int, std::string> > getIssuerAlternativeNames() const = 0;

        virtual DistinguishedName getSubjectDN() const = 0;
        virtual std::vector<std::pair<int, std::string> > getSubjectAlternativeNames() const = 0;

        virtual int getVersion() const = 0;
        virtual std::string toString() const = 0;

        virtual std::vector<std::shared_ptr<X509Extension>> getX509Extensions() const = 0;
        virtual std::shared_ptr<X509Extension> getX509Extension(const std::string&) const = 0;

        virtual std::vector<Ice::Byte> getAuthorityKeyIdentifier() const = 0;
        virtual std::vector<Ice::Byte> getSubjectKeyIdentifier() const = 0;

        static std::shared_ptr<Certificate> load(const std::string&);
        static std::shared_ptr<Certificate> decode(const std::string&);
    };
}
```

C++98

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

        virtual bool verify(const CertificatePtr&) const = 0;
        virtual std::string encode() const = 0;
        virtual bool checkValidity() const = 0;

        virtual bool checkValidity(const IceUtil::Time&) const = 0;

        virtual IceUtil::Time getNotAfter() const = 0;
        virtual IceUtil::Time getNotBefore() const = 0;

        virtual std::string getSerialNumber() const = 0;

        virtual DistinguishedName getIssuerDN() const = 0;
        virtual std::vector<std::pair<int, std::string> > getIssuerAlternativeNames() const = 0;

        virtual DistinguishedName getSubjectDN() const = 0;
        virtual std::vector<std::pair<int, std::string> > getSubjectAlternativeNames() const = 0;

        virtual int getVersion() const = 0;
        virtual std::string toString() const = 0;

        virtual std::vector<X509ExtensionPtr> getX509Extensions() const = 0;
        virtual X509ExtensionPtr getX509Extension(const std::string&) const = 0;

        virtual std::vector<Ice::Byte> getAuthorityKeyIdentifier() const = 0;
        virtual std::vector<Ice::Byte> getSubjectKeyIdentifier() const = 0;

        static CertificatePtr load(const std::string&);
        static CertificatePtr decode(const std::string&);
    };
}
```

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

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

> **UWP**
>
> The UWP implementation of load uses GetFileFromApplicationUriAsync and the file path must use an appropriate URI format.

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.

Both `load` and `decode` return a certificate using the default `IceSSL::Certificate` implementation:

- `IceSSL::SChannel::Certificate` on Windows
- `IceSSL::SecureTransport::Certificate` on macOS
- `IceSSL::UWP::Certificate` for UWP
- `IceSSL::OpenSSL::Certificate` for OpenSSL

All of these classes derive from `IceSSL::Certificate` to provide platform-dependent methods.

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

The `checkValidity` functions 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 a `system_clock::time_point` with the C++11 mapping (`IceUtil::Time` with the C++98 mapping) and returns true if the certificate is valid at the given time.

The `getNotAfter` and `getNotBefore` functions return the times that define the certificate's valid period, as a `system_clock::time_point` with the C++11 mapping and as a `IceUtil::Time` with the C++98 mapping.

The functions `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 functions return instances of the class `IceSSL::DistinguishedName`, another convenience class that is described below.

The `getX509Extensions` and `getX509Extension` functions can be used to retrieve a list of the X509 extensions included in the certificate or a specific X509 extension by providing its `OID`, respectively. These methods are only supported by the OpenSSL and SChannel implementations. We discuss extensions further below.

The `getAuthorityKeyIdentifier` and `getSubjectKeyIdentifier` functions return the authority key identifier and subject key identifier respectively as a sequence of bytes. These methods are not supported with UWP or with SecureTransport on iOS.

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

Back to Top ^

Interacting with the Native Certificates

It is possible to interact with the SSL implementation's native certificates. Tthese APIs are platform-dependent and are defined in extensions of the `IceSSL::Certificate` class. Refer to the header file for each implementation to get more details. The following table summarizes the header files specific to each implementation:

SSL Implementation	Header File
OpenSSL	`IceSSL/OpenSSL.h`
SecureTransport	`IceSSL/SecureTransport.h`
SChannel	`IceSSL/SChannel.h`
UWP	`IceSSL/UWP.h`

Usually you don't need to include these header files in your application; you'll just include `IceSSL/IceSSL.h` which includes the appropriate header files for your platform's default implementation. If you are using the OpenSSL implementation on Windows and need to access OpenSSL-specific APIs, you must include `IceSSL/OpenSSL.h`.

Back to Top ^

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:

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

        bool match(const DistinguishedName&) const;
        bool match(const std::string&) const;

        operator std::string() const;

        friend bool operator==(const DistinguishedName&, const DistinguishedName&);
        friend bool operator<(const DistinguishedName&, const DistinguishedName&);
    };

    inline bool operator!=(const DistinguishedName&, const DistinguishedName&) { ... }
    // etc., provides all comparison operators.
}
```

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 `match` functions perform a partial comparison that does not consider the order of relative distinguished names. Given two instances N1 and N2 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.

The comparison operators 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.

Back to Top ^

Using Certificate Extensions in C++

The `Certificate` class provides the methods `getX509Extensions` and `getX509Extension` for obtaining information about its X509 extensions.

> These methods are only supported when using the OpenSSL or Schannel implementations.

The methods return instances of `IceSSL::X509Extension`:

C++11

```
namespace IceSSL
{
    class X509Extension
    {
    public:

        virtual bool isCritical() const = 0;
        virtual std::string getOID() const = 0;
        virtual std::vector<Ice::Byte> getData() const = 0;
    };
}
```

C++98

```
namespace IceSSL
{
    class X509Extension : public virtual IceUtil::Shared
    {
    public:

        virtual bool isCritical() const = 0;
        virtual std::string getOID() const = 0;
        virtual std::vector<Ice::Byte> getData() const = 0;
    };
}
```

An OID is represented as a string. For example, the OID for the subject alternative name extension is `"2.5.29.17"`. The data associated with the extension is provided as a vector of bytes.

The following code demonstrates how to retrieve an extension:

C++11

```
shared_ptr<IceSSL::Certificate> cert = ...;
shared_ptr<IceSSL::X509Extension> subjectAltName = cert->getX509Extension("2.5.29.17");
if(subjectAltName)
{
    vector<Ice::Byte> data = subjectAltName->getData();
    ...
}
```

C++98

```
IceSSL::CertificatePtr cert = ...;
IceSSL::X509ExtensionPtr subjectAltName = cert->getX509Extension("2.5.29.17");
if(subjectAltName)
{
    vector<Ice::Byte> data = subjectAltName->getData();
    ...
}
```

Back to Top ^

See Also

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

Previous

Next