Configuring IceSSL

After installing IceSSL, an application typically needs to define a handful of additional properties to configure settings such as the location of certificate and key files. This page provides an introduction to configuring the plug-in for each of the supported language mappings.

On this page:

- C++ Configuration for IceSSL
  - DSA Example for C++
  - RSA and DSA Example for C++
  - ADH Example for C++
- Java Configuration for IceSSL
  - DSA Example for Java
  - ADH Example for Java
- .NET Configuration for IceSSL
  - Managing Certificates with the Microsoft Management Console
  - Using Certificate Files
  - Using Certificate Stores
  - Importing Certificates
- Ice Touch Configuration for IceSSL
  - Keychains
- Configuring Ciphersuites
  - Configuring Ciphersuites in C++
  - Configuring Ciphersuites in Java
- Configuring Trust Relationships
  - Trusted Peers
  - Verification Depth
- Configuring Secure Proxies
- IceSSL Diagnostics
  - System Logging in Java
  - System Logging in .NET

C++ Configuration for IceSSL

Our first example shows the properties that are sufficient in many situations:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=pubkey.pem
IceSSL.KeyFile=privkey.pem
IceSSL.CertAuthFile=ca.pem
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your certificate and key files. The three properties that follow it define the files containing the program's certificate, private key, and trusted CA certificate, respectively. This example assumes the files contain RSA keys, and IceSSL requires the files to use the Privacy Enhanced Mail (PEM) encoding. Finally, the `IceSSL.Password` property specifies the password of the private key.

Note that it is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

DSA Example for C++

If you used DSA to generate your keys, one additional property is necessary:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=pubkey_dsa.pem
IceSSL.KeyFile=privkey_dsa.pem
IceSSL.CertAuthFile=ca.pem
IceSSL.Password=password
IceSSL.Ciphers=DEFAULT:DSS
```

The `IceSSL.Ciphers` property adds support for DSS authentication to the plug-in's default set of ciphersuites.

RSA and DSA Example for C++

It is also possible to specify certificates and keys for both RSA and DSA by including two filenames in the `IceSSL.CertFile` and `IceSSL.KeyFile` properties. The filenames must be separated using the platform's path separator. The example below demonstrates the Unix separator (a colon):

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=pubkey_rsa.pem:pubkey_dsa.pem
IceSSL.KeyFile=privkey_rsa.pem:privkey_dsa.pem
IceSSL.CertAuthFile=ca.pem
IceSSL.Password=password
IceSSL.Ciphers=DEFAULT:DSS
```

On Windows, you would use a semicolon to separate the filenames.

ADH Example for C++

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.Ciphers=ADH
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH, which is disabled by default.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

Java Configuration for IceSSL

IceSSL uses Java's native format for storing keys and certificates: the keystore.

A keystore is represented as a file containing key pairs and associated certificates, and is usually administered using the `keytool` utility supplied with the Java run time. Keystores serve two roles in Java's SSL architecture:

1. A keystore containing a key pair identifies the peer and is usually closely guarded.
2. A keystore containing public certificates represents the identities of trusted peers and can be freely shared. These keystores are also referred to as "truststores" when they are used to store only trusted certificate chains.

A single keystore file can fulfill both of these purposes.

Java supports a pluggable architecture for keystore implementations in which a system property selects a particular implementation as the default keystore type. IceSSL uses the default keystore type unless otherwise specified.

A password is assigned to each key pair in a keystore, as well as to the keystore itself. IceSSL must be provided with the password for the key pair, but the keystore password is optional. If a keystore password is specified, it is used only to verify the keystore's integrity. IceSSL requires that all of the key pairs in a keystore have the same password.

Our first example shows the properties that are sufficient in many situations:

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Keystore=keys.jks
IceSSL.Truststore=ca.jks
IceSSL.Password=password
```

IceSSL resolves the filenames defined in its configuration properties as follows:

1. Attempt to open the file as a class loader resource. This is especially useful for deploying applications with special security restrictions, such as applets.
2. Attempt to open the file in the local file system.
3. If `IceSSL.DefaultDir` is defined, prepend its value and try steps 1 and 2 again. The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your keystore and truststore files.

The `IceSSL.Password` property specifies the password of the key pair. Note that it is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

DSA Example for Java

Java supports both RSA and DSA keys. No additional properties are necessary to use DSA:

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Keystore=dsakeys.jks
IceSSL.Truststore=ca.jks
IceSSL.Password=password
```

ADH Example for Java

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Ciphers=NONE (DH_anon)
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH, which is disabled by default.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

.NET Configuration for IceSSL

The Common Language Runtime (CLR) in .NET uses certificate stores as the persistent repositories of certificates and keys. Furthermore, the CLR maintains two distinct sets of certificate stores, one for the current user and another for the local machine. Although it is possible to load a certificate and its corresponding private key from a regular file, the CLR requires trusted CA certificates to reside in an appropriate certificate store.

Managing Certificates with the Microsoft Management Console

On Windows, you can use the Microsoft Management Console (MMC) to browse the contents of the various certificate stores. To start the console, run `MMC.EXE` from a command window, or choose Run from the Start menu and enter `MMC.EXE`.

Once the console is running, you need to install the Certificates "snap-in" by choosing Add/Remove Snap-in from the File menu. Click the Add button, choose Certificates in the popup window and click Add. If you wish to manage certificates for the current user, select My Current Account and click Finish. To manage certificates for the local computer, select Computer Account and click Next, then select Local Computer and click Finish.

When you have finished adding snap-ins, close the Add Standalone Snap-in window and click OK on the Add/Remove Snap-in window. Your Console Root window now contains a tree structure that you can expand to view the available certificate stores. If you have a certificate in a file that you want to add to a store, click on the desired store, then open the Action menu and select All Tasks/Import.

Using Certificate Files

Our first example demonstrates how to configure IceSSL with a file that contains the program's certificate and key:

```
Ice.Plugin.IceSSL=IceSSL.dll:IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=cert.pfx
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your certificate file. This file must use the Personal Information Exchange (PFX, also known as PKCS#12) format and contain both a certificate and its corresponding private key. The `IceSSL.Password` property specifies the password used to secure the file.

Note that it is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

This configuration assumes that any trusted CA certificates necessary to authenticate the program's peers are already installed in an appropriate certificate store. You may also use a configuration property to automatically import a certificate from a file, as described in below.

Using Certificate Stores

If the program's certificate and private key are already installed in a certificate store, you can select it using the `IceSSL.FindCert` configuration property as shown in the following example:

```
Ice.Plugin.IceSSL=IceSSL.dll:IceSSL.PluginFactory
IceSSL.FindCert.LocalMachine.My=subject:"Quote Server"
```

An `IceSSL.FindCert` property executes a query in a particular certificate store and selects all of the certificates that match the given criteria. In the example above, the location of the certificate store is `LocalMachine`, and the store's name is `My`. When using MMC to browse the certificate stores, this specification is equivalent to the store "Personal" in the location "Certificates (Local Computer)."

The other legal value for the location component of the property name is `CurrentUser`. The following table shows the valid values for the store name component and their equivalents in MMC.

Property Name	MMC Name
`AddressBook`	Other People
`AuthRoot`	Third-Party Root Certification Authorities
`CertificateAuthority`	Intermediate Certification Authorities
`Disallowed`	Untrusted Certificates
`My`	Personal
`Root`	Trusted Root Certification Authorities
`TrustedPeople`	Trusted People
`TrustedPublisher`	Trusted Publishers

The search criteria consists of *name:value* pairs that perform case-insensitive comparisons against the fields of each certificate in the specified store, and the special property value `*` selects every certificate in the store. Typically, however, the criteria should select a single certificate. In a server, IceSSL must supply the CLR with the certificate that represents the server's identity; if a configuration matches several certificates, IceSSL chooses one (in an undefined manner) and logs a warning to notify you of the situation.

Selecting a certificate from a store is more secure than using a certificate file via the `IceSSL.CertFile` property because it is not necessary to specify a plain-text password. MMC prompts you for the password when initially importing a certificate into a store, so the password is not required when an application uses that certificate to identify itself.

Importing Certificates

IceSSL can be configured to import a certificate into a particular store. The Ice demos and test suites use this capability to ensure that the CA certificate is present, which avoids the need for a user to manually import the certificate using MMC before using the Ice sample programs and tests.

The `IceSSL.ImportCert` property uses the same format for its name as the `IceSSL.FindCert` property described above, in that the certificate store's location and name are part of the property name:

```
IceSSL.ImportCert.LocalMachine.AuthRoot=cacert.pem
```

The property's value is the name of a certificate file and an optional password. If a file is protected with a password, append the password to the property value using a semicolon as the separator. IceSSL uses the value of `IceSSL.DefaultDir` to complete the file name if necessary. The CLR accepts a number of encoding formats for the certificate, including PEM, DER and PFX.

The store name should be chosen with care. When installing a trusted CA certificate, authentication succeeds only when the certificate is installed into one of the following stores:

- `LocalMachine.Root`
- `LocalMachine.AuthRoot`
- `CurrentUser.Root`

Note that administrative privileges are required when installing a certificate into a `LocalMachine` store.

If you specify a store name other than those listed in the table above, IceSSL creates a new store with the given name and adds the certificate to it. Once installed in the specified store, the application (or the user) is responsible for removing the certificate when it is no longer necessary.

Ice Touch Configuration for IceSSL

In Ice Touch, certificate files are loaded from the application's resource bundle. If the application's target platform is Mac OS, certificate files can also be loaded directly from the file system. Consider the following properties:

```
IceSSL.DefaultDir=certs
IceSSL.CertAuthFile=cacert.der
IceSSL.CertFile=cert.pfx
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the location of your certificate files. Defining `IceSSL.DefaultDir` means IceSSL searches for certificate files relative to the specified directory. For the properties in the example above, IceSSL composes the pathnames `certs/cacert.der` and `certs/cert.pfx`. If `IceSSL.DefaultDir` is not defined, IceSSL uses the certificate file pathnames exactly as they are supplied.

As mentioned earlier, IceSSL has different semantics for locating certificate files depending on the target platform. For the iPhone and iPhone simulator, IceSSL attempts to open a certificate file in the application's resource bundle as `Resources/`*`DefaultDir/file`* if `IceSSL.DefaultDir` is defined, or as simply `Resources/`*`file`* otherwise. If the target platform is Mac OS and the certificate file cannot be found in the resource bundle, IceSSL also attempts to open the file in the file system as *`DefaultDir/file`* if a default directory is specified, or as simply *`file`* otherwise.

IceSSL requires that the CA certificate file specified by `IceSSL.CertAuthFile` use the DER format. The certificate file in `IceSSL.CertFile` must use the Personal Information Exchange (PFX, also known as PKCS#12) format and contain both a certificate and its corresponding private key. The `IceSSL.Password` property specifies the password used to secure the certificate file.

Keychains

IceSSL imports the certificate specified by `IceSSL.CertFile` into a keychain. IceSSL uses the `login` keychain by default unless you choose a different one by defining the `IceSSL.Keychain` property:

```
IceSSL.Keychain=Test Keychain
```

The `login` keychain is the user's default keychain, which is normally unlocked after logging into the system. IceSSL does not usually require a password to import a certificate into the `login` keychain. However, if your `login` keychain is not unlocked automatically, or if you have selected a different keychain, you can supply a password using the `IceSSL.KeychainPassword` property:

```
IceSSL.KeychainPassword=password
```

Configuring Ciphersuites

A ciphersuite represents a particular combination of encryption, authentication and hashing algorithms. The IceSSL plug-ins for C++ and Java allow you to configure the ciphersuites that their underlying SSL engines are allowed to negotiate during handshaking with a peer. By default, IceSSL uses the underlying engine's default ciphersuites, but you can define a property to customize the list as we demonstrated above with the ADH examples. Normally the default configuration is chosen to eliminate relatively insecure ciphersuites such as ADH, which is the reason it must be explicitly enabled.

Configuring Ciphersuites in C++

The value of the `IceSSL.Ciphers` property is given directly to the low-level OpenSSL library, on which IceSSL is based. Therefore, OpenSSL determines the allowable ciphersuites, which in turn depend on how the OpenSSL distribution was compiled. You can obtain a complete list of the supported ciphersuites using the `openssl` command:

```
$ openssl ciphers
```

This command will likely generate a long list. To simplify the selection process, OpenSSL supports several classes of ciphers, as shown in the following table.

Class	Description
ALL	All possible combinations.
ADH	Anonymous ciphers.
LOW	Low bit-strength ciphers.
EXP	Export-crippled ciphers.

Classes and ciphers can be excluded by prefixing them with an exclamation point. The special keyword `@STRENGTH` sorts the cipher list in order of their strength, so that SSL gives preference to the more secure ciphers when negotiating a cipher suite. The `@STRENGTH` keyword must be the last element in the list.

For example, here is a good value for the `IceSSL.Ciphers` property:

```
ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH
```

This value excludes the ciphers with low bit strength and known problems, and orders the remaining ciphers according to their strength.

Note that no warning is given if an unrecognized cipher is specified.

Configuring Ciphersuites in Java

IceSSL for Java interprets the value of `IceSSL.Ciphers` as a sequence of expressions that filter the selected ciphersuites using name and pattern matching. If the property is not defined, the Java security provider's default ciphersuites are used. The following table defines the valid expressions that may appear in the property value.

Expression	Description
`NONE`	Disables all ciphersuites. If specified, it must appear first.
`ALL`	Enables all supported ciphersuites. If specified, it must appear first. This expression should be used with caution, as it may enable low-security ciphersuites.
`NAME`	Enables the ciphersuite matching the given name.
`!`*`NAME`*	Disables the ciphersuite matching the given name.
`(EXP)`	Enables ciphersuites whose names contain the regular expression *EXP*.
`!(EXP)`	Disables ciphersuites whose names contain the regular expression *EXP*.

To determine the set of enabled ciphersuites, the plug-in begins with a list of ciphersuite names containing the default set as determined by the security provider. The expressions in the property value add and remove ciphersuites from this list and are evaluated in the order of appearance. For example, consider the following property definition:

```
IceSSL.Ciphers=NONE (RSA.*AES) !(EXPORT)
```

The expressions in this property have the following effects:

- `NONE` clears the list of enabled ciphersuites.
- `(RSA.*AES)` is a regular expression that enables ciphersuites whose names contain the string "RSA" followed by "AES", meaning ciphersuites using RSA authentication and AES encryption.
- `!(EXPORT)` is a regular expression that disables any of the selected ciphersuites whose names contain the string "EXPORT", meaning ciphersuites having export-quality strength.

As another example, this property adds anonymous Diffie-Hellman to the default set of ciphersuites and disables export ciphersuites:

```
IceSSL.Client.Ciphers=(DH_anon) !(EXPORT)
```

Finally, this example selects only one ciphersuite:

```
IceSSL.Client.Ciphers=NONE SSL_RSA_WITH_RC4_128_SHA
```

Configuring Trust Relationships

Declaring that you trust a certificate authority implies that you trust any peer whose certificate was signed directly or indirectly by that certificate authority. It is necessary to use this broad definition of trust in some applications, such as a public Web server. In more controlled environments, it is a good idea to restrict access as much as possible, and IceSSL provides a number of ways for you to do that.

Trusted Peers

After the low-level SSL engine has completed its authentication process, IceSSL can be configured to take additional steps to verify whether a peer should be trusted. The `IceSSL.TrustOnly` family of properties defines a collection of acceptance and rejection filters that IceSSL applies to the distinguished name of a peer's certificate in order to determine whether to allow the connection to proceed. IceSSL permits the connection if the peer's distinguished name matches any of the acceptance filters and does not match any of the rejection filters.

A distinguished name uniquely identifies a person or entity and is generally represented in the following textual form:

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

Suppose we are configuring a client to communicate with the server whose distinguished name is shown above. If we know that the client is allowed to communicate only with this server, we can enforce this rule using the following property:

```
IceSSL.TrustOnly=O="ZeroC, Inc.", OU=Servers, CN=Quote Server
```

With this property in place, IceSSL allows a connection to proceed only if the distinguished name in the server's certificate matches this filter. The property may contain multiple filters, separated by semicolons, if the client needs to communicate with more than one server. Additional variations of the property are also supported.

If the `IceSSL.TrustOnly` properties do not provide the selectivity you require, the next step is to install a custom certificate verifier.

Verification Depth

In order to authenticate a peer, SSL obtains the peer's certificate chain, which includes the peer's certificate as well as that of the root CA. SSL verifies that each certificate in the chain is valid, but there still remains a subtle security risk. Suppose that we have identified a trusted root CA (via its certificate), and a peer has supplied a valid certificate chain signed by our trusted root CA. It is possible for an attacker to obtain a special signing certificate that is signed by our root CA and therefore trusted implicitly. The attacker can use this certificate to sign fraudulent certificates with the goal of masquerading as a trusted peer, presumably for some nefarious purpose.

We could use the `IceSSL.TrustOnly` properties described above in an attempt to defend against such an attack. However, the attacker could easily manufacture a certificate containing a distinguished name that satisfies the trust properties.

If you know that all trusted peers present certificate chains of a certain length, set the property `IceSSL.VerifyDepthMax` so that IceSSL automatically rejects longer chains. The default value of this property is two, therefore you may need to set it to a larger value if you expect peers to present longer chains.

In situations where you cannot make assumptions about the length of a peer's certificate chain, yet you still want to examine the chain before allowing the connection, you should install a custom certificate verifier.

Configuring Secure Proxies

Proxies may contain any combination of secure and insecure endpoints. An application that requires secure communication can guarantee that proxies it manufactures itself, such as those created by calling `stringToProxy`, contain only secure endpoints. However, the application cannot make the same assumption about proxies received as the result of a remote invocation.

The simplest way to guarantee that all proxies use only secure endpoints is to define the `Ice.Override.Secure` configuration property:

```
Ice.Override.Secure=1
```

Setting this property is equivalent to invoking the proxy method `ice_secure(true)` on every proxy. When enabled, attempting to establish a connection using a proxy that does not contain a secure endpoint results in `NoEndpointException`.

If you want the default behavior of proxies to give precedence to secure endpoints, you can set this property instead:

```
Ice.Default.PreferSecure=1
```

Note that proxies may still attempt to establish connections to insecure endpoints, but they try all secure endpoints first. This is equivalent to invoking `ice_preferSecure(true)` on a proxy.

IceSSL Diagnostics

You can use two configuration properties to obtain more information about the plug-in's activities. Setting `IceSSL.Trace.Security=1` enables the plug-in's diagnostic output, which includes a variety of messages regarding events such as ciphersuite selection, peer verification and trust evaluation. The other property, `Ice.Trace.Network`, determines how much information is logged about network events such as connections and packets. Note that the output generated by `Ice.Trace.Network` also includes other transports such as TCP and UDP.

System Logging in Java

In Java, you can use a system property that displays a great deal of information about SSL certificates and connections, including the ciphersuite that is selected for use by each connection. For example, the following command sets the system property that activates the diagnostics:

```
$ java -Djavax.net.debug=ssl MyProgram
```

System Logging in .NET

Enabling additional tracing output in .NET requires the creation of an XML file such as the one shown below:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <system.diagnostics>
        <trace autoflush="true"/>
        <sources>
            <source name="System.Net">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
            <source name="System.Net.Sockets">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
            <source name="System.Net.Cache">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
        </sources>
        <sharedListeners>
            <add
            name="System.Net"
            type="System.Diagnostics.TextWriterTraceListener"
            initializeData="trace.txt"
            />
        </sharedListeners>
        <switches>
            <add name="System.Net" value="Verbose"/>
            <add name="System.Net.Sockets" value="Verbose"/>
            <add name="System.Net.Cache" value="Verbose"/>
        </switches>
    </system.diagnostics>
</configuration>
```

In this example, the output is stored in the file `trace.txt`. To activate tracing, give the XML file the same name as your executable with a `.config` extension (such as `server.exe.config`), and place it in the same directory as the executable.

See Also

- Proxy Methods
- Filtering Proxy Endpoints
- Public Key Infrastructure
- Using IceSSL
- Programming IceSSL
- Advanced IceSSL Topics
- IceSSL Properties