Connections
ZeroC's Newsletter for the Ice Community

Issue Number 17, September 2006

Tower of Babel

Recently, a close friend of mine (who is also a programmer) mentioned that "there are far too many programming languages". If found myself agreeing with him. There are literally thousands of programming languages. In fact, we have more *programming* languages than *human* ones. The Encyclopedia of Computer Languages currently lists 8,512 programming languages. For comparison, Ethnologue records 6,912 human languages. (Of these, only 215 languages have more than one million speakers, and about 2,000 have fewer than 1,000 speakers.)

It seems obvious that most of the 8,512 programming languages are unnecessary, in the sense that they do not provide functionality that is not also available in other languages. (And, in contrast to human languages, it is hard to argue that a programming language has intrinsic cultural value). One could even argue that we need only *one* programming language because, ultimately, all programming languages are Turing-complete, and therefore equivalent.

The equivalence of programming languages struck me recently while I was translating code from C# to Visual Basic. .NET provides some rather nice language integration features. For example, you can derive a VB class from a base class that is implemented in C# or Managed C++. To make this possible, .NET's Common Language Infrastructure (CLI) imposes a Common Type System (CTS) on all languages that produce managed code. In practice, this means that these languages must agree on a common set of built-in types, must agree on a common object model, must agree on a common syntax for identifiers, and must agree on a whole host of other things. While this enables seamless language integration, it also creates languages that are all based on a lowest common denominator. The choice of programming language becomes almost arbitrary: whether you write something in C#, VB, or Managed C++ is largely a matter of choosing which syntax you like best. Of course, this begs the question: why is it that we have two or three programming languages when a single one would do?

Another (and more pertinent) question is "Why do we have C#?" After all, Java was around quite some time before C# and, for all intents and purposes, C# is almost indistinguishable from Java. Of course, the answer can be found in history: Microsoft wanted to modify Java to conform to the CLI, and Sun disagreed with that. And the industry was bestowed with a new programming language that it needed about as much as it needed a hole in the head. As a result, dozens of tool providers created hundreds of compilers, debuggers, IDEs, syntax-directed editors, browsers, and other tools for C#; thousands of programmers had to learn C#; dozens of authors wrote books about C#; hundreds of magazines and web sites published thousands of articles about C#; thousands of developers spent tens of thousands of hours attending training courses on C#; and thousands of programmers rewrote millions of lines of code in C#. The total cost of all this to the industry (and, ultimately, to consumers) is easily in the multi-hundred million dollar range. All this for a new programming language that added essentially nothing...

Another (and more contentious) question is "How many programming languages do we really need?" Being fully aware of the thin ice (pun intended) I am skating on, I believe that, for general-purpose commercial programming, we need only four (yes, *four*) languages. Here they are: we need one low-level and efficient system programming language that is close to the actual hardware: C++; we need one general-purpose application development language that is type-safe and can run in managed execution environments: Java, C#, Eiffel, or similar; we need one general-purpose scripting language for ad-hoc development, prototyping, and knocking up small(-ish) tools: Perl, Python, Ruby, or similar; and we need a database query language: SQL. That's it: four languages can cover something like 95% of all commercial software development.

Of course, I'm being facetious here; there is ample justification for other languages such as Scheme, Prolog, Mathematica, VHDL, assembly language, and so on. But I'm worried by the height of this tower of Babel: 8,512 stories are far too many for comfort; I'd be happier with a baker's dozen or so. Ice currently supports C++, Java, C#, VB, Python, PHP, and—soon to come—Ruby. A rather modest tower by comparison, but one that is roomy enough for the majority of software developers. Oh yes, one more thing… Did I mention that variety is the spice of life? Enjoy, and bon appétit!

Michi

Michi Henning
Chief Scientist

Issue Features

IceGrid Security

Matthew Newhook describes in detail how to secure IceGrid.

Contents

IceGrid Security

Matthew Newhook, Senior Software Engineer

Introduction

In the previous article, we created an MP3 encoder that uses the IceGrid session allocation mechanism to control client access to the encoding resources. In this article, I will show how we can secure the grid against unauthorized access. There are quite a few ways to secure the grid; two common configurations are Glacier2 in conjunction with a firewall, and SSL. This article demonstrates both.

There are two separate (but related) concerns we need to discuss to secure the grid. The first is how to authenticate end-users, that is, clients that use the grid, and the second is how to protect the grid from unauthorized access.

We can authenticate clients either via a user name and password, or via the credentials associated with a connection. User name and password authentication requires an implementation of the `Glacier2::PermissionsVerifier` interface, such as the file-based permissions verifier provided by IceGrid and Glacier2, or a custom implementation of your own. Authentication via credentials requires a secure connection. Specifically, it relies on the X509 certificate chain associated with an SSL connection. For this method of authentication, you need to provide an implementation of the `Glacier2::SSLPermissionsVerifier` interface. (Note that, to implement this interface, your code must link with the Glacier2 libraries, but need not use a Glacier2 router.)

Either method of authentication requires a secure connection to avoid eavesdropping attacks because, without SSL, the user name and password would be sent over the network in plain text. In addition, the client should always authenticate the server-side through a certificate. Without such authentication, the client cannot know whether it is talking to the correct server and, with user name and password authentication, could easily be duped into providing its credentials to an attacker. Note that eavesdropping is not a problem for authentication via connection credentials because the certificate's private key is not sent over the wire (unlike the password in case of user name and password authentication).

The second concern is how to protect the grid itself, that is, how to prevent unauthorized users from accessing grid resources. For example, if a client somehow got hold of a proxy to a grid resource (perhaps an application saved a previously-allocated proxy), the grid must prevent the client from accessing this resource. Protecting the grid requires using Glacier2 or SSL (or potentially both).

The most common type of deployment for an organization that wants to allow access to grid resources from outside the corporate network is to use a firewall in conjunction with Glacier2. The core strategy is to firewall the entire server back end and allow client access only via Glacier2. All clients are required to authenticate with Glacier2 before accessing grid resources, and direct access to the server back end is restricted to system administrators only. (Whether you need any further protection for the server back end depends on your installation.)

You can also secure a grid with SSL only. However, doing this is far more complex because it requires a thorough understanding of the relationships between the various IceGrid components. I will describe these relationships in detail and show how to configure IceGrid to restrict communications to only those components that need it. This type of fully-restricted installation is more onerous to administer and, for reasons I will explain later, does not support user name and password authentication, so the more common approach is to use Glacier2 and a firewall.

The IceSSL Plug-in

Before we further discuss how to protect IceGrid, we should review the IceSSL plug-in. The plug-in provides two separate but equally important functions:

- Encryption. This protects communications from eavesdropping. (However, depending on how the plug-in is configured, communications may still be subject to man-in-the-middle attacks. In particular, anonymous Diffie-Hellman suffers from this vulnerability—see http://en.wikipedia.org/wiki/Diffie-Hellman for details.)
- Authentication. This establishes the digital identity of one or both of the communicating parties.

X509 Certificates

The IceSSL plug-in establishes the digital identity of a party using an X509 digital certificate. An X509 certificate consists of two parts: the certificate itself, and the certificate's private key. Trust comes from the fact that a party cannot claim to be the owner of a certificate without its private key. The private key must be kept secret by its owner because anyone who gains access to the private key can impersonate the true owner. The private key is typically stored in encrypted form so it cannot be used without knowledge of a password.

Certificates are generally composed into a sequence called a *chain*, with each certificate signed by the next certificate in the chain. The signing is done in a secure manner—it is not possible to sign a certificate without the private key of the certificate of the signer (and that private key is, of course, kept secure). Typically, certificates consist of a two-element chain. The beginning (or *root*) of the chain is known as a *certificate authority* (*CA*). A CA that signs a certificate vouches for the owner of the certificate: if you trust a CA to do its job correctly then, given a certificate signed by the CA, you can be sure that the owner of that certificate is who he or she claims to be.

Exactly what information is contained within an X509 certificate? Let us take a look at the header of an X509 certificate:

```
$ head -35 registry_cert.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=Grid CA, O=GridCA-may.local/ema
ilAddress=matthew@zeroc.com
        Validity
            Not Before: Aug 10 02:30:32 2006 GMT
            Not After : Aug  9 02:30:32 2011 GMT
        Subject: CN=IceGrid Registry, O=GridCA-may
.local
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
               ....
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Key Identifier:
                B3:DA:86:CA:A2:AD:BF:5E:54:
CF:26:01:B1:97:4B:55:35:11:14:A7
            X509v3 Authority Key Identifier:
                keyid:CD:1B:63:47:17:BF:D8:67:88:5
D:D6:CA:FA:2A:8F:50:FE:02:80:8C
                DirName:/CN=Grid CA/O=GridCA-may.l
ocal/emailAddress=matthew@zeroc.com
                serial:D3:3A:85:B6:37:33:02:AF

    Signature Algorithm: md5WithRSAEncryption
```

As you can see, there is quite a bit of information here. If you want the full details, you can read the X509 certificate specification, however, the most important things in this certificate are the *distinguished name*, or *DN*, of the issuer of the certificate (our CA, in this case), and the DN of the certificate. The other items are important, but are checked automatically by the SSL toolkit, and so usually do not need to be examined by your application code.

What exactly is a DN? A DN is a name that uniquely identifies an entry. It consists of a number of `attribute=value` pairs separated by commas. Each pair is called a *relative distinguished name*, or *RDN*. The order of RDNs is significant because each DN represents a path from a root directory to the level where the entry resides. Let's look at the DN of the certificate:

```
CN=IceGrid Registry, O=GridCA-may.local
```

This DN has two RDNs. `CN`, which is the *common name*, and `O`, which is used to represent the *organization*. They have the values `IceGrid Registry` and `GridCA-may.local`, respectively.

There are many commercial CA's that you can use. However, Ice provides a CA implementation that is geared towards use with IceGrid. This implementation is a wrapper around functionality already offered by OpenSSL.

To use the Ice CA, you need to follow these steps:

- Initialize the CA with the `initca.py` script. This creates a root CA certificate.
- Create certificate requests with the `req.py` script. You can use these for the IceGrid registry, IceGrid nodes, Ice servers, and individual users.
- Sign the certificate requests with the `sign.py` script to turn them into signed certificates.
- If you want to use the certificates with Java or C#, use the `import.py` script to perform the necessary conversions between the C++ PEM format and the C# and Java formats.

See the Ice manual for more information on these scripts.

Authorization

Authentication is essential for authorization. Authentication is the act of establishing the digital identity of the parties, whereas authorization is the act of determining exactly what activities the parties may participate in. The SSL plug-in provides a number of separate mechanisms for authorization. By default, the SSL plug-in allows unlimited communications for any party that provides a certificate satisfying the following constraints:

- The root CA certificate is self-signed and among the application's trusted CA certificates.
- All other certificates in the chain are signed by the one immediately preceding it. (By default, the certificate chain must have length of 2).
- None of the certificates have expired.

It is possible to further restrict the accepted set of certificates by using the `TrustOnly` series of properties. These properties configure a set of filters that are applied to the distinguished name of a peer's certificate in order to determine whether to accept a connection. (You can also implement custom policies by implementing the `IceSSL::CertificateVerifier` interface—see the Ice manual for more information).

The `TrustOnly` properties allow you to easily configure both the client and the server roles to only trust a limited set of DNs:

- `IceSSL.TrustOnly`. This property limits both client and server communications to those peers that match the property value.
- `IceSSL.TrustOnly.Client`. This property limits client-side connections to only those endpoints that match the property value.
- `IceSSL.TrustOnly.Server`. This property limits all server endpoints to accept connections only from those peers that match the property value.
- `IceSSL.TrustOnly.Server.<AdapterName>`. This property limits the named object adapter to accept connections only from those peers that match the property value.

The property value is a list of RDNs that must be matched without consideration of order. Alternative matches are separated with a '`;`' character. For example, to match the DN shown above (`CN=IceGrid Registry, O=GridCA-may.local`), the following would work:

```
CN=IceGrid Registry
```

As would this:

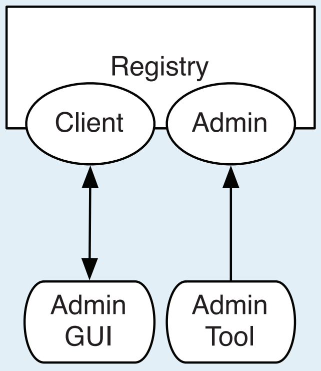
```
O=GridCA-may.local, CN=IceGrid Registry
```

Since, for this article, we control the only CA, we need only specify the common name (`CN` RDN attribute) for matches. If you use a third-party CA, you will likely have to match on all of its DN elements to ensure that you do not get an accidental match. (This is certainly safest and, unless the CA is strictly controlled, this is the recommended approach.)

Securing Administrative Access

An important topic that we must address is administration of the IceGrid registry. It is very important to protect access to the registry because its deployment mechanism can be used to launch arbitrary executables on the grid. There are currently two tools that you can use to administer the registry: the `icegridadmin` command-line tool, and the Java-based IceGrid GUI. At present, these tools use two different authentication methods.

Figure 1: Registry Endpoints for the `icegridadmin` and GUI tools



The `icegridadmin` tool accesses the registry via the Admin endpoint(s). You can secure these endpoints by enabling only SSL (but not TCP), and by configuring the trust rules to only trust a given set of user certificates.

The IceGrid GUI uses a different authentication mechanism. It first establishes an administrative session via the `IceGrid::Registry` interface:

```
// Slice
module IceGrid
{

interface Registry
{
    // ...
    AdminSession*
    createAdminSession(
        string userId, string password)
        throws PermissionDeniedException;
    AdminSession*
    createAdminSessionFromSecureConnection()
        throws PermissionDeniedException;
};

};
```

As for client sessions, there are two alternative methods to create an administrative session: by providing a user name and password, or by using the credentials associated with a secure connection. Because administrative access is potentially dangerous, you should never permit it over an insecure connection. Unfortunately, as we just saw, the IceGrid GUI uses the registry's client endpoints, which are the same endpoints that are used by clients to create client sessions. This means if you want to use TCP for client sessions and SSL for administrative sessions, you have a potential security hole because an administrator could accidentally use the TCP endpoint and send user name and password over an insecure connection. To prevent this potential problem, you must require administrative sessions to be authenticated only via a secure connection. (A future release of IceGrid will rectify this shortcoming.)

Administrative session creation uses its own set of permission verifiers. For user name and password authentication, IceGrid uses the permission verifier defined by the property `IceGrid.Registry.AdminPermissionsVerifier` whereas, for secure connection authentication, IceGrid uses the SSL permissions verifier defined by the property `IceGrid.Registry.AdminSSLPermissionsVerifier`. To permit authentication only via secure connections, you must set only the `AdminSSLPermissionsVerifier` property, but not the `AdminPermissionsVerifier` property.

SSL Permissions Verifier

We'll look at how to configure the IceGrid registry to permit administrative access with the GUI only via a secure endpoint. The following is a brief summary of what you must do:

- Implement the `Glacier2::SSLPermissionsVerifier` interface and configure a proxy to an instance of this interface by setting the `IceGrid.Registry.AdminSSLPermissionsVerifier` property in the IceGrid registry configuration file.
- Generate a certificate each for the registry and the permissions verifier, and at least one certificate for the administrative user.
- Import the administrative certificate into a Java key store, using the `import.py` script.

Let's look at the Slice definition of the `SSLPermissionsVerifier` interface:

```
// Slice
module Glacier2
{

struct SSLInfo
{
    string remoteHost;
    int remotePort;
    string localHost;
    int localPort;
    string cipher;
    StringSeq certs;
};

interface SSLPermissionsVerifier
{
    nonmutating bool
    authorize(SSLInfo info, out string reason);
};

};
```

Your implementation of this object must examine the information in the `SSLInfo` object to determine whether or not to trust the peer. In theory, it can use all of the information in this structure to determine whether or not to authorize access. However, the most important piece of information is the certificate chain. When IceGrid calls your `authorize` method, the IceSSL plug-in has already validated the chain to make sure that certificates have not expired, are properly signed, and so on, so your application code does not need to repeat this process.

Up to this point, I have only discussed direct authentication with the IceGrid registry. However, the IceGrid GUI and IceGrid are also capable of authenticating via a Glacier2 router. The configuration is very similar, except that you must configure the verifiers on the Glacier2 router instead of the registry, so I will not cover this configuration in detail (see Figure 2 and Figure 3). I will provide more information on Glacier2 and IceGrid configuration later in this article.

Figure 2: Direct Authentication with the Registry

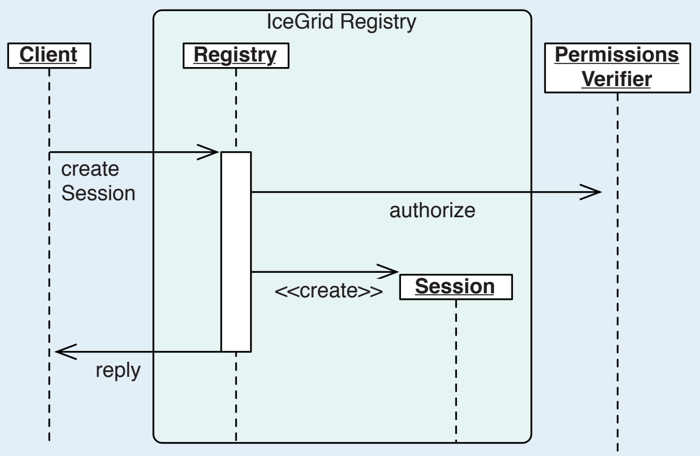
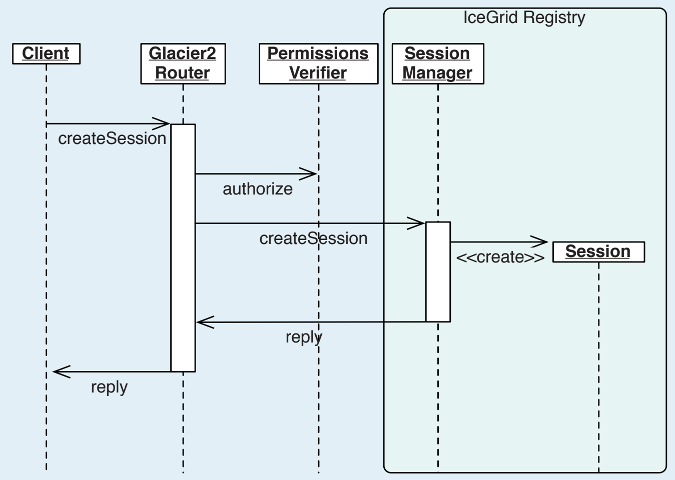


Figure 3: Authentication with the Registry via Glacier2



The contents of a certificate chain are a sequence of *Privacy Enhanced Mail* (*PEM*) encoded certificates. In C++, the `decode` method of the `IceSSL::Certificate` class creates a certificate from its PEM encoding. (See the Ice manual for details on how to this with C# and Java.) Once you have decoded the certificate, you typically examine its DN to decide whether or not to authorize the user. For example, you might look up the DN in a database or an LDAP server—your implementation is free to use whatever is appropriate.

Our example implementation is very simple: it compares the certificate DN with a specific required DN:

```
// C++
class SSLPermissionsVerifierI :
    public Glacier2::SSLPermissionsVerifier
{
public:
    SSLPermissionsVerifierI() :
        _dn("emailAddress=matthew@zeroc.com,"
            "O=GridCA-may.local,CN=IceGrid Admin")
    {
    }
    virtual bool
    authorize(const Glacier2::SSLInfo& info,
              string&, const Current&) const
    {
        if(info.certs.size() > 1)
        {
            IceSSL::CertificatePtr cert =
                IceSSL::Certificate::decode(
                    info.certs[0]);
            if(_dn == cert->getSubjectDN())
            {
                return true;
            }
        }
        return false;
    }
private:
    const DistinguishedName _dn;
};
```

Note that this code uses the IceSSL `DistinguishedName` class to compare the DNs, instead of using a straight string comparison. The `DistinguishedName` class takes care of many details that you would otherwise have to handle correctly when comparing DNs, such as character escapes, white space rules, and so on. (If you are interested in these rules, you can read RFC 2253—Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names.) The Ice manual provides more detail on the `DistinguishedName` class.

Where do we host our permissions verifier object? For now, to simplify the configuration, we will host the object in a stand-alone server. I will explain shortly how to use IceGrid to deploy that server. The implementation of the server is as follows:

```
// C++
class PermissionsVerifierServer : public
Application
{
public:

    virtual int run(int, char*[])
    {
        ObjectAdapterPtr adapter =
            communicator()->createObjectAdapter(
                "PermissionsVerifier");
        adapter->add(new SSLPermissionsVerifierI,
            communicator()->stringToIdentity(
                "AdminSSLPermissionsVerifier"));
        adapter->activate();
        communicator()->waitForShutdown();
        return EXIT_SUCCESS;
    }
};

int
main(int argc, char* argv[])
{
    PermissionsVerifierServer app;
    return app.main(argc, argv,
        "config.verifier");
}
```

What about the configuration for this verifier? Does it need to be protected? The answer depends on the IceGrid configuration. Assuming that the verifier is not protected by a firewall, we must protect it ourselves using SSL. Before we can protect anything though, we need to generate some certificates. (The demo distribution that accompanies this article contains all the necessary certificates, together with a log of exactly how they were generated.)

To generate certificates, you must first initialize your certificate authority:

```
$ initca.py
This script will initialize your organization's
Certificate Authority (CA).
The CA database will be created in /Users/
matthew/.iceca/ca
The subject name for your CA will be
CN=Grid CA ,  O=GridCA-may.local
Do you want to keep this as the CA subject name?
(y/n) [y]y
Enter the email address of the CA: matthew@zeroc.c
om
Generating configuration files...  ca.cnf  sign.
cnf req.cnf ok
Generating a 2048 bit RSA private key
......+++
.................................+++
writing new private key to '/Users/matthew/.iceca/
ca/db/ca_key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----

The CA is initialized.

You need to distribute the following files to all
machines that can
request certificates:

/Users/matthew/.iceca/req.cnf
/Users/matthew/.iceca/ca_cert.pem

These files should be placed in the user's home di
rectory in
~/.iceca. On Windows, place these files in <ice-in
stall>/config.
```

You need to copy the file `~/.iceca/ca_cert.pem` into the demo source directory, so that the CA certificate is accessible to the grid.

Next we'll create the registry certificate:

```
$ req.py --registry
Generating a 1024 bit RSA private key
...........++++++
......................++++++
writing new private key to 'registry_key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----

Created key: registry_key.pem
Created certificate request: registry_req.pem

The certificate request must be signed by the CA.
Send the certificate
request file to the CA at the following email addr
ess:
matthew@zeroc.com
```

After you have created the certificate request, it must be signed by the CA. For this demo we can sign the request immediately since we *are* the CA:

```
$ sign.py --in registry_req.pem --out registry_cer
t.pem
Using configuration from /Users/matthew/.iceca/ca/
sign.cnf
Enter pass phrase for /Users/matthew/.iceca/ca/db/
ca_key.pem:
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
organizationName      :PRINTABLE:'GridCA-may.
local'
commonName            :PRINTABLE:'IceGrid
Registry'
Certificate is to be certified until Aug  1
03:34:30 2011 GMT (1825 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit?
[y/n]y
Write out database with 1 new entries
Data Base Updated
```

Now we'll change the IceGrid configuration to use these certificates with the SSL plug-in. The first step is to add the SSL base configuration information to the IceGrid registry. The certificates are assumed to live in the directory `certs`.

```
# IceGrid Registry Configuration
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
IceSSL.CertFile=registry_cert.pem
IceSSL.KeyFile=registry_key.pem
IceSSL.DefaultDir=certs
```

Next, generate a new server certificate (use the `-server` option to `req.py`) for the verifier (use the name `SSL Verifier`). This certificate must be stored in `verifier_cert.pem` and `verifier_key.pem`. Then add the SSL settings to the permissions verifier application configuration:

```
# Permissions Verifier Configuration
PermissionsVerifier.Endpoints=ssl -p 11112
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
IceSSL.CertFile=verifier_cert.pem
IceSSL.KeyFile=verifier_key.pem
IceSSL.DefaultDir=certs
```

You must provide a proxy to this verifier object in the registry's configuration:

```
# IceGrid Registry Configuration
IceGrid.Registry.AdminSSLPermissionsVerifier=Admin
SSLPermissionsVerifier:ssl -p 11112
```

You must configure the SSL verifier application to trust only the IceGrid registry—with this configuration, the application will not communicate with any server or client that does not have the given common name.

```
# SSL Verifier configuration
IceSSL.TrustOnly=CN="IceGrid Registry"
```

You may also want to add a trust rule to the IceGrid registry to the client-side trust only set, as follows:

```
# IceGrid Registry Configuration
IceSSL.TrustOnly.Client=CN="Ice Server SSL Verifie
r"
```

Next, create a certificate for the IceGrid GUI administrative user with the `–user` option to `req.py`. The user name must be "`IceGrid Admin`" and the e-mail address must be matthew@zeroc.com. (Of course, in reality you would generate whatever makes sense for your environment, but for the purposes of this demo we need a well-defined common name and e-mail address). This generates a DN that matches the string we provided above. The certificate must be stored in `admin_cert.pem` and `admin_key.pem`.

Since the IceGrid GUI is a Java application, the certificates must be converted to a form suitable for Java. Fortunately, the simple CA included with Ice contains just such a script—`import.py`. This script imports a certificate into a Java key store.

```
$ import.py --java admincert admin_cert.pem admin_
key.pem adminstore.jks
Enter private key passphrase:
Enter keystore password:
converting to pkcs12 format...  ok
importing into the keystore... ok
```

Now you need to create a configuration file for the IceGrid GUI. The configuration file must load the SSL plug-in. (If you do not do this, the GUI will get an `EndpointParseException` when it attempts to connect to the registry.)

```
# IceGrid GUI Configuration File
Ice.Plugin.IceSSL=IceSSL.PluginFactory
```

In addition, the GUI should have a trust rule so that it only trusts the IceGrid registry. That way, administrators can be sure that they are connecting to the correct registry.

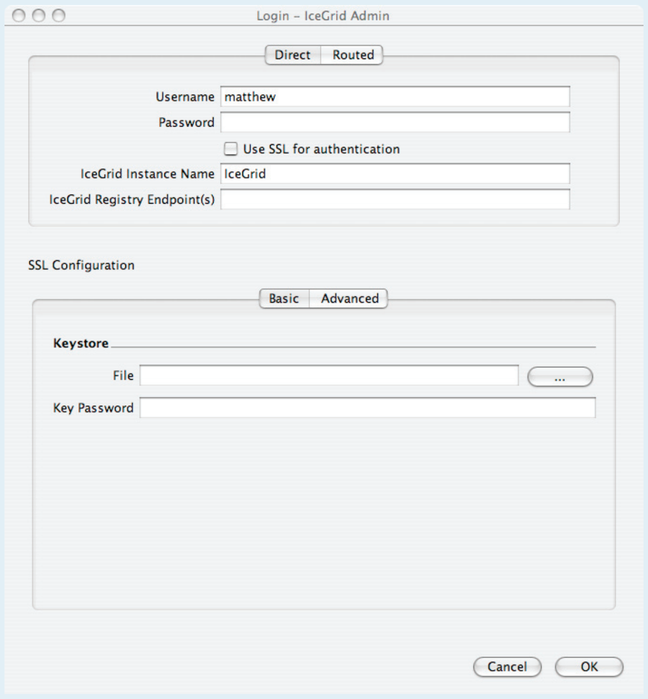
```
# IceGrid GUI Configuration File
IceSSL.TrustOnly=CN="IceGrid Registry"
```

Now, start the GUI with this configuration file.

```
$ java -jar /opt/Ice-3.1.0/lib/IceGridGUI.jar --Ic
e.Config=config.gui
```

After selecting "`login`", you will be presented with a dialog as shown in Figure 4.

Figure 4: IceGrid GUI Login Dialog


Login – IceGrid Admin

Direct | Routed

Username: matthew

Password:

☐ Use SSL for authentication

IceGrid Instance Name: IceGrid

IceGrid Registry Endpoint(s):

SSL Configuration

Basic | Advanced

Keystore

File:

Key Password:

You must follow these steps:

- Check “Use SSL for authentication”.
- Change “IceGrid Instance Name” to `EncoderIceGrid`.
- Enter `ssl -p 12100` for the “IceGrid Registry Endpoints”. (Obviously, if you are administering remotely, you must add the host on which the registry runs with `-h hostname`.)
- Select the “Advanced” tab in “SSL Configuration”.
- Select the `adminstore.jks` key store.
- Select `admincert` in the drop-down list. This is necessary as the key store created by the `import.py` script contains two certificates, the admin certificate and the CA certificate.

The configuration should look something like Figure 5:

Press OK, and you will be logged into the registry via a secure connection.

SSL

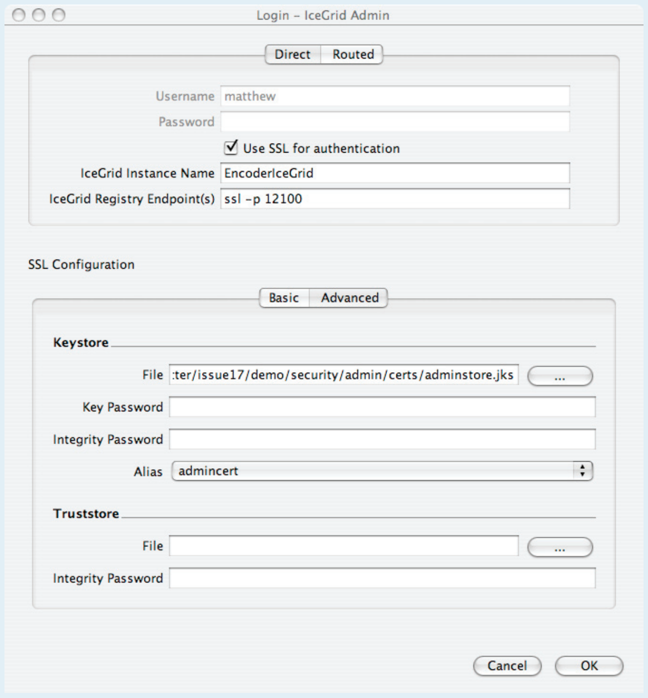
In this configuration, we will secure the entire grid using SSL only, without the help of a firewall or Glacier2. This configuration fully protects the IceGrid registry and nodes, as well as access to the Ice-Grid resources by clients. Clients have direct access to all internal resources. I want to emphasize here that this is one possible setup for IceGrid with SSL protection—you do not have to do it in this way. For example, you might want to firewall off portions of the network to have a safer, simpler, or less attack-prone setup.

First, a brief summary of the steps you must take to secure the grid using SSL only. This summary assumes that you have a working deployed grid application, and now you want to protect it using SSL using one certificate for all nodes, and one certificate for all grid services.

- Generate certificates for the registry, nodes, grid services, servers, SSL permissions verifier, and administrators, plus a certificate for each grid user.
- Write an implementation of the `Glacier2::SSLPermissionsVerifier`. Configure SSL endpoints for the server, and define the trust relationships so that only the registry can connect to that server.
- Configure SSL endpoints on the registry and each of the nodes in the grid. (Do not configure TCP endpoints). Configure the trust rules between the registry and the nodes.
- Configure SSL endpoints for each grid service in the deployment, and any servers used by the grid services. (Once again, no TCP endpoints are permitted.) Define the correct trust rules.
- Configure each allocatable grid service to use `session` activation.
- Configure SSL for the client-side application, and configure the trust rules appropriately.

Now we will examine the various components that we have in the deployment and the relationships that they will establish with their peers.

Figure 5: IceGrid GUI Login Configuration


Login – IceGrid Admin

Direct | Routed

Username: matthew

Password:

☑ Use SSL for authentication

IceGrid Instance Name: EncoderIceGrid

IceGrid Registry Endpoint(s): ssl -p 12100

SSL Configuration

Basic | Advanced

Keystore

File: :ter/issue17/demo/security/admin/certs/adminstore.jks

Key Password:

Integrity Password:

Alias: admincert

Truststore

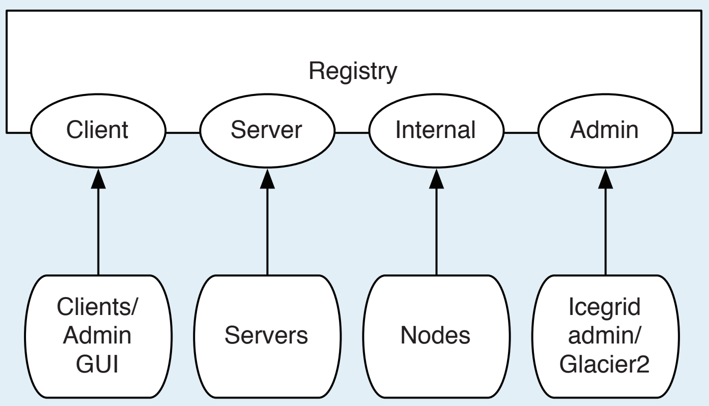
File:

Integrity Password:

IceGrid Components

Firstly, let's look at the most complex component, the IceGrid registry.

Figure 6: IceGrid Registry

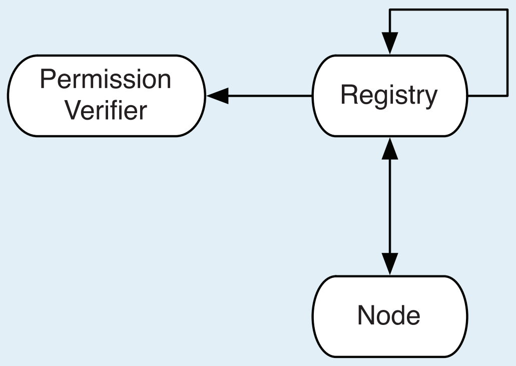


The ovals represent the available endpoints in the IceGrid registry. We'll go through each in turn:

- Client. This endpoint supports the `IceGrid::Query`, `IceGrid::Locator`, and `IceGrid::Registry` interfaces. It must be accessible to any application that uses IceGrid. This endpoint is also used by the IceGrid GUI application.
- Server. Only servers that host indirect object adapters use this endpoint.
- Internal. Only IceGrid nodes use this endpoint.
- Admin. The `icegridadmin` command-line tool uses this endpoint. Additionally it is used by Glacier2 to create sessions.

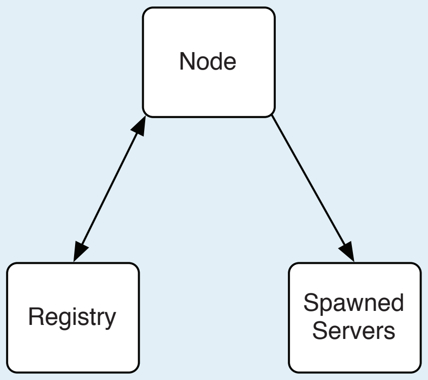
Next we look at the components with which the IceGrid registry communicates.

Figure 7: IceGrid Registry



The IceGrid registry communicates directly with each IceGrid node, with any configured permission verifier objects, and with itself. Next, the IceGrid node:

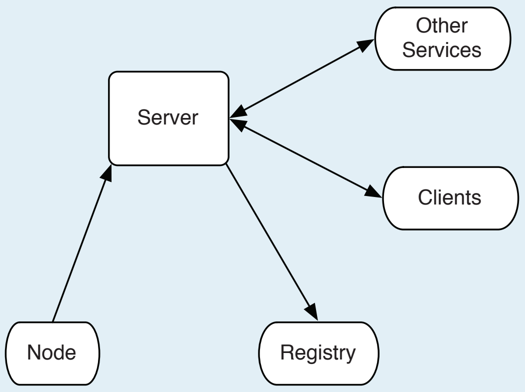
Figure 8: IceGrid Node



Each IceGrid node accepts connections from the registry and sends update information to the registry. Each node also communicates with the servers it spawns.

Next, we examine a grid service. This is the meat of your application—the services that the grid provides to its clients. We will first examine a generic service, and then look specifically at the encoder factory service.

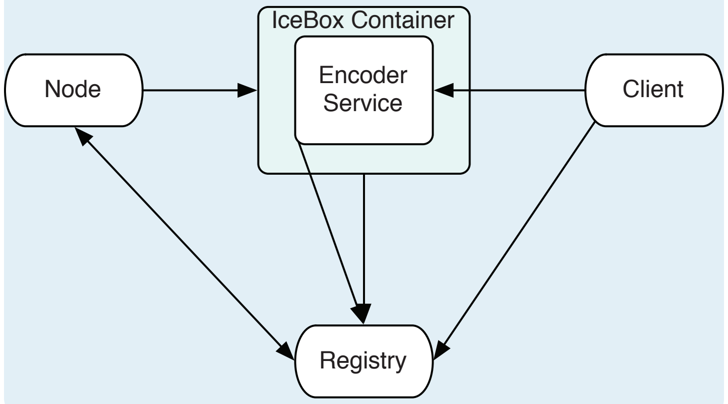
Figure 9: IceGrid Service



Here we have an Ice grid service. The IceGrid node must be able to talk to the process object in the spawned server so it can tell it when to shut down. (Technically, the node connects to the endpoint hosted by the object adapter that hosts the process object—see the Ice manual for more information). Each indirect object adapter registers its endpoints with the IceGrid registry during activation. (This is the case for any IceGrid service.) To determine the remainder of the relationships you need to examine exactly how your service is used. For example, services will allow uni- or bidirectional communication with IceGrid clients and, often, some IceGrid services will use (or will be used) by other IceGrid services or servers on the network. You need to examine your application in detail to work out what these relationships are.

For a concrete example, let us look at the encoder factory service.

Figure 10: Encoder Factory Service



Since the encoder factory is hosted inside an IceBox container, the deployed server actually has two components that must be suitably configured. Firstly, we have the IceBox service container. The IceBox container must accept connections from the IceGrid node. Furthermore, the service communicates with the IceGrid registry in order to register its `Process` object. Secondly, we have the encoder factory IceBox service. That services communicates with the IceGrid registry and must accept connections from clients.

Certificates

Now that we know the relationships between the various services, we can think about how to protect them. Here are the certificates we need to generate:

- One certificate for the IceGrid registry.
- One certificate for all of the IceGrid nodes. (Note that we could generate a separate certificate for each node, but that would require a restart of the IceGrid registry whenever we add a new node.)
- One certificate for all IceGrid services. This is quite coarse-grained because it does not allow us to distinguish among the different IceGrid servers available in the grid. A more fine-grained approach would generate a separate certificate for each service but that is more complex, of course: for every service we add, we would need to edit the registry configuration, as well as update the node configuration on each node on which the server is deployed to describe the relationships between the servers in the server's configuration files.
- One certificate for all of the admin users. (Of course, it is also possible to have a separate certificate for each admin user, at the cost of more complex configuration.)
- One certificate for each user of the encoding service.

We'll create certificates with the Ice simple CA package. I assume that you have initialized the CA, and that the registry is configured as described previously. (In the examples that follow, each of the certificates must be signed. However, I have not shown the signing step since it is identical to what I showed earlier.)

First, we create the node certificate. The node name should be `All`—this will result in the common name `IceGrid Node All`. The node certificate and private key must be placed into files called `node_cert.pem` and `node_key.pem`.

```
$ req.py --node
Enter the node name: All
Generating a 1024 bit RSA private key
................++++++
..................++++++
writing new private key to 'node_key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----

Created key: node_key.pem
Created certificate request: node_req.pem

The certificate request must be signed by the CA.
Send the certificate
request file to the CA at the following email add
ress:
matthew@zeroc.com
```

Next, we generate the server certificate. In contrast to the registry and node certificates, server certificates must generally not be password protected. If a server's certificate were password protected, the IceGrid node could not start the server (because it would need the password to do that). The alternative, namely setting `IceSSL.Password`, would mean embedding the password in plain text in the deployment file. I believe that it is better to use no password, and instead protect the certificate's private key via appropriate operating system permissions. The server name should be `All`, which will result in the common name `Ice Server All`. The certificates must be placed into files called `gridserver_cert.pem` and `gridserver_key.pem`.

```
$ req.py --server --no-password
Enter the server name: All
Generating a 1024 bit RSA private key
.......++++++
.........++++++
writing new private key to 'server_key.pem'
-----

Created key: server_key.pem
Created certificate request: server_req.pem

The certificate request must be signed by the CA.
Send the certificate
request file to the CA at the following email add
ress:
matthew@zeroc.com
```

In addition, you should re-generate the SSL permission verifier certificate without password protection (since IceGrid will manage this service). Next, you need to generate a user certificate for use by the encoder client. For example:

```
$ req.py --user
Enter the user's full name: Matthew Newhook
Enter the user's email address: matthew@zeroc.com
Generating a 1024 bit RSA private key
...++++++
..++++++
writing new private key to 'user_key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----

Created key: user_key.pem
Created certificate request: user_req.pem

The certificate request must be signed by the CA.
Send the certificate
request file to the CA at the following email add
ress:
matthew@zeroc.com
```

Trust Relationships

For outgoing connections, the registry must trust only the IceGrid node. For implementation reasons, the registry also connects to itself and therefore must trust itself. We can configure this as follows:

```
# IceGrid registry configuration
IceSSL.TrustOnly.Client=CN="IceGrid Registry";CN="
IceGrid Node All"
```

As previously described, the registry has four sets of endpoints. The client endpoints do not have any configured trust rules—access to this endpoint is permitted only for those clients who are authenticated by one of the configured permissions verifiers. We'll run through the remainder in turn. The admin endpoints can only be accessed by the admin user—in this case that is the user with the CN `IceGrid Admin`.

```
# IceGrid registry configuration
IceSSL.TrustOnly.Server.IceGrid.Registry.Admin=CN=
"IceGrid Admin"
```

The server endpoints must be accessible to IceGrid services:

```
# IceGrid registry configuration
IceSSL.TrustOnly.Server.IceGrid.Registry.Server=CN
="Ice Server All"
```

Finally, the internal endpoints must accessible to nodes and the registry itself:

```
# IceGrid registry configuration
IceSSL.TrustOnly.Server.IceGrid.Registry.Internal=
CN="IceGrid Node All";CN="IceGrid Registry"
```

The endpoints must be configured to only use SSL, as follows:

```
# IceGrid registry configuration
IceGrid.Registry.Client.Endpoints=ssl -p 12000
IceGrid.Registry.Server.Endpoints=ssl
IceGrid.Registry.Internal.Endpoints=ssl
IceGrid.Registry.Admin.Endpoints=ssl
```

Next, we look at the IceGrid node configuration. Again, we need to add the base SSL configuration information:

```
# IceGrid node configuration
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
IceSSL.CertFile=node_cert.pem
IceSSL.KeyFile=node_key.pem
IceSSL.DefaultDir=certs
```

We also need to describe the trust relationships. On the client side, the node talks with the IceGrid registry and spawned IceGrid services:

```
# IceGrid node configuration
IceSSL.TrustOnly.Client=CN="Ice Server All";CN="Ic
eGrid Registry"
```

Since the IceGrid node only has one object adapter, we can use the `TrustOnly.Server` property to specify the trust relationship (instead of the more specific property that specifies the object adapter name). The node only accepts connections from the IceGrid registry:

```
# IceGrid node configuration
IceSSL.TrustOnly.Server=CN=”IceGrid Registry”
```

Now we can start the registry and the node. If you have problems starting these processes, you can set the security tracing property to help you isolate the source of the problem:

```
IceSSL.Trace.Security=1
```

Next, we must modify the deployment. Note that the encoder factory is deployed within an IceBox container. Therefore, we must configure both the encoder factory and the container. First, we'll add the base SSL configuration to the container:

```
// IceGrid deployment descriptor
<icebox id="${instance-name}" exe="icebox"
activation="session">
  <property name="IceBox.ServiceManager.Endpoints"
   value="ssl -h ${host}"/>
  <property name="Ice.Plugin.IceSSL"
   value="IceSSL:createIceSSL"/>
  <property name="IceSSL.CertAuthFile"
   value="ca_cert.pem"/>
  <property name="IceSSL.CertFile"
   value="gridserver_cert.pem"/>
  <property name="IceSSL.KeyFile"
   value="gridserver_key.pem"/>
  <property name="IceSSL.DefaultDir"
   value="certs"/>
```

Note that this descriptor specifies session activation mode. That mode is necessary to properly secure the encoder factory. (We'll examine the reason for this in a moment.)

Next, we define the trust rules. On the client side, the container must trust the registry and, on the server side, it must trust the IceGrid node so the node can use the `Process` object. Because the container has only one object adapter, we can again use the generic `TrustOnly.Server` property:

```
# IceGrid deployment descriptor
<property name="IceSSL.TrustOnly.Client"
 value='CN="IceGrid Registry"'/>
<property
 name="IceSSL.TrustOnly.Server.IceBox.ServiceManag
er"
 value='CN="IceGrid Node All"'/>
```

Next, we examine the configuration for the encoder factory service. Again, we need to add the base SSL configuration:

```
# IceGrid deployment descriptor
<service name="${instance-name}"
 entry="Mp3EncoderService:create">
  <property name="Ice.Plugin.IceSSL"
   value="IceSSL:createIceSSL"/>
  <property name="IceSSL.CertAuthFile"
   value="ca_cert.pem"/>
  <property name="IceSSL.CertFile"
   value="gridserver_cert.pem"/>
  <property name="IceSSL.KeyFile"
   value="gridserver_key.pem"/>
  <property name="IceSSL.DefaultDir"
   value="certs"/>
```

For client-side invocations, we only need to trust the registry:

```
# IceGrid deployment descriptor
<property name="IceSSL.TrustOnly.Client"
 value='CN="IceGrid Registry"'/>
```

Now, we need to look at the server-side trust rules. The encoder factory must trust client applications, but only those clients that have allocated a factory. (A client application that has not allocated a factory must not be able to talk to the server.) This cannot be expressed via static configuration but we can use IceGrid's `session.id` substitution variable to achieve what we want. The variable contains the value of the client session identifier and, for sessions created from a secure connection, it contains the DN of the user's credentials. Thus, we can use the following property:

```
<property name="IceSSL.TrustOnly.Server"
 value='${session.id}'/>
```

Since session mode activation means that the server is re-started each time a server is allocated to a user, the variable always contains the DN of the user. For example, my user certificate DN is as follows:

```
CN=Matthew Newhook, O=GridCA-may.local/emailAddres
s=matthew@zeroc.com
```

Once my client has allocated a server, this property is expanded to:

```
IceSSL.TrustOnly.Server=CN=Matthew Newhook, O=Grid
CA-may.local/emailAddress=matthew@zeroc.com
```

This gives us exactly what we need: only the client with these credentials can access the encoder factory endpoints, and all other clients are rejected. This also explains why we cannot use user name and password for authentication if we secure the entire grid with SSL: in that case, the `session.id` variable would contain the user name instead of the DN and would expand to:

```
IceSSL.TrustOnly.Server=matthew
```

This clearly would not be workable but, without a trust rule, any client could access the server (not only the client that allocated the server), meaning that it would not be correctly protected.

The remaining task is to change the adapter endpoints to SSL:

```
# IceGrid deployment descriptor
<adapter name="${instance-name}-EncoderFactory"
 endpoints="ssl -h ${host}">
  <allocatable
   identity="${instance-name}-Mp3Encoder"
   type="::Ripper::Mp3EncoderFactory"/>
</adapter>
```

Now that we have a deployment descriptor, we need to write a configuration file for `icegridadmin` so we can deploy the descriptor. Since `icegridadmin` needs to communicate with the registry only, the trust rule is very simple:

```
# IceGrid admin configuration file
IceGrid.InstanceName=EncoderIceGrid
Ice.Default.Locator=EncoderIceGrid/Locator:ssl -p
12000
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
IceSSL.CertFile=admin_cert.pem
IceSSL.KeyFile=admin_key.pem
IceSSL.DefaultDir=certs
IceSSL.TrustOnly=CN="IceGrid Registry"
```

Now we move to the client configuration. The base SSL configuration is as follows:

```
# Client configuration file
Ice.Default.Locator=EncoderIceGrid/Locator:ssl -p
12000
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
IceSSL.CertFile=matthew_cert.pem
IceSSL.KeyFile=matthew_key.pem
IceSSL.DefaultDir=certs
```

The client must trust the IceGrid registry and IceGrid services, which leads to the following rule:

```
# Client configuration file
IceSSL.TrustOnly=CN="IceGrid Registry";CN="Ice Ser
ver All"
```

SSL Session Creation

We need a few code changes to establish the session via a secure connection instead of using user name and password authentication. (If the client were to continue to use user name and password authentication, its connection would be rejected—even if an accidentally left-behind permissions verifier were to accept the user name and password—because the trust rule would be violated.) The code for the session creation is straightforward:

```
// C++
IceGrid::RegistryPrx registry = IceGrid::
    RegistryPrx::checkedCast(
        communicator()->stringToProxy(
            "EncoderIceGrid/Registry"));
IceGrid::SessionPrx session;
try
{
    session =
    registry->createSessionFromSecureConnection();
}
catch(const IceGrid::PermissionDeniedException&ex)
{
    cout << "permission denied:\n" << ex.reason
         << endl;
    return 1;
}
```

If we try this client, however, we will get an error:

```
$ ./client testcase.wav
...
permission denied:
no ssl permissions verifier configured, use the
property `IceGrid.Registry.SSLPermissionsVerifier'
to configure a permissions verifier.
```

What did we forget to do? As the error says, there is no permissions verifier configured to authorize SSL connections. Unfortunately, there is no equivalent of the `NullPermissionsVerifier` for SSL available with Ice 3.1, so we will have to write it ourselves. Our implementation is very simple. (The actual source code is not quite as shown but has the same semantics.)

```
// C++
class SSLPermissionsVerifierI :
    public Glacier2::SSLPermissionsVerifier
{
public:
    virtual bool
    authorize(const Glacier2::SSLInfo&,
              string&, const Current&) const
    {
        return true;
    }
};
```

We'll host this object along with the existing admin SSL permissions verifier with the identity `SSLPermissionsVerifier`. Note that hosting the admin permissions verifier with IceGrid presents a bootstrapping problem if you want to use the IceGrid GUI: the problem is that in order to load the initial configuration that contains the admin permissions verifier, you need to run the IceGrid GUI—however, authenticating the tool requires the as-yet undeployed permissions verifier. To get around this, you can either use the `-deploy` option to `icegridregistry` to load the initial deployment, temporarily use the null permissions verifier, or use the `icegridadmin` command line tool (which, as of Ice 3.1, doesn't use the same authentication mechanism).

Let's look at the deployment descriptor:

```
# IceGrid Deployment Descriptor
<server id="verifier" exe="../services/
sslverifier" activation="on-demand">
  <property name="IceSSL.CertAuthFile"
   value="ca_cert.pem"/>
  <property name="IceSSL.CertFile"
   value="verifier_cert.pem"/>
  <property name="IceSSL.KeyFile"
   value="verifier_key.pem"/>
  <property name="IceSSL.DefaultDir"
   value="certs"/>
  <adapter name="PermissionsVerifier"
   endpoints="ssl -h localhost">
    <object identity="SSLPermissionsVerifier"/>
    <object
     identity="AdminSSLPermissionsVerifier"/>
  </adapter>
</server>
```

We need to set the SSL permissions verifier property and the trust rules so that the IceGrid registry can talk to the permissions verifier:

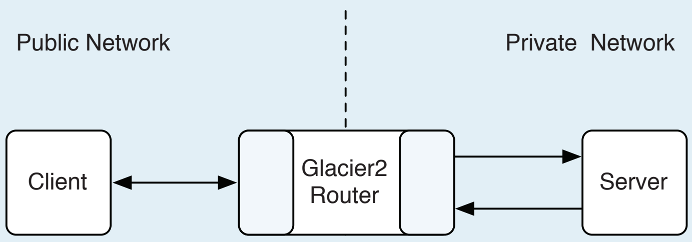
```
# IceGrid Registry Configuration
IceSSL.TrustOnly.Client=CN="IceGrid Registry";CN="
IceGrid Node All";CN="Ice Server SSL Verifier"
IceSSL.TrustOnly.Server.IceGrid.Registry.Server=CN
="Ice Server All";CN="Ice Server SSL Verifier"
IceGrid.Registry.SSLPermissionsVerifier=SSLPermiss
ionsVerifier
```

It is worth remembering that, while this configuration is quite complex, it is very secure (provided that no-one can access the private certificate keys). Also note that this is only one of many possible SSL certificate setups. More restrictive (and complex) setups are also possible. For example, you might want to have different classes of servers, some which cannot be accessed by particular clients.

Glacier2

Now we will examine a setup using Glacier2 in conjunction with a firewall (which is a common configuration):

Figure 11: Glacier2



Here all machines inside the network are trusted and the grid itself is secured by the firewall. Glacier2 takes care of securing access to grid resources. In other words, all of the IceGrid administrative tools have direct access to the IceGrid nodes and registry because we assume no malicious code will run inside the firewall. Clients access IceGrid via Glacier2.

Our initial setup will use TCP between Glacier2 and the client, and will use user name and password authentication. As I mentioned previously, this is not safe because it sends the user name and password over the wire in clear text. However, we'll first look at this simple configuration and then modify it for use with SSL. A summary of the steps involved in using Glacier2 to protect the grid is as follows:

- Modify the IceGrid deployment to deploy a Glacier2 router.
- Modify the client configuration to define a default router instead of a default locator.
- Modify the client code to establish the IceGrid session with the Glacier2 router instead of with the IceGrid registry.
- Modify the deployment descriptor for the allocatable objects to use server or session allocation instead of object allocation.

First, we will modify the deployment descriptor to deploy Glacier2. We will use the predefined Glacier2 template that accompanies the Ice distribution, so we need to tell the descriptor to import the default templates as follows:

```
# IceGrid Deployment Descriptor
<icegrid>
  <application name="Mp3Ripper"
   import-default-templates="true">
```

Next, we deploy Glacier2.

```
# IceGrid Deployment Descriptor
<server-instance template="Glacier2"
 client-endpoints="tcp -h 1.2.3.4 -p 10005"
 server-endpoints="tcp"
 session-timeout="60"/>
```

Glacier2 clients access Glacier2 on the public IP address `1.2.3.4` at port `10005`. Server endpoints use a random port on their local host. We also need to tell Glacier2 to use the IceGrid session manager and permissions verifier implementation. Note that the session manager object is hosted on the admin endpoints in the IceGrid registry. Since we are using only TCP in this configuration it is sufficient to ensure that the admin endpoints are enabled. (With SSL, we would have to add suitable trust rules to the admin endpoint for the Glacier2 router.)

```
# IceGrid Deployment Descriptor
<server-instance template="Glacier2" ...>
 <properties>
  <property name="Glacier2.SessionManager"
   value="EncoderIceGrid/SessionManager"/>
  <property name="Glacier2.PermissionsVerifier"
   value="EncoderIceGrid/NullPermissionsVerifier"
/>
 </properties>
</server-instance>
```

Next, we need to modify the client application to create the session with Glacier2 instead of the IceGrid registry:

```
// C++
RouterPrx defaultRouter =
    communicator()->getDefaultRouter();
if(!defaultRouter)
{
    cerr << argv[0] << ": no default router set"
         << endl;
    return EXIT_FAILURE;
}

Glacier2::RouterPrx router =
    Glacier2::RouterPrx::checkedCast(
        defaultRouter);
if(!router)
{
    cerr << argv[0]
         << ": configured router is not a "
         << " Glacier2 router" << endl;
    return EXIT_FAILURE;
}
IceGrid::SessionPrx session;
try
{
    session = IceGrid::SessionPrx::uncheckedCast(
        router->createSession("foo", "bar"));
}
catch(const IceGrid::PermissionDeniedException&ex)
{
    cout << "permission denied:\n" << ex.reason
         << endl;
    return 1;
}
```

This code is almost identical to the code for creating a session with the IceGrid registry, except that the router comes from configuration. The session refresh timeout interval also comes from the Glacier2 router:

```
// C++
SessionRefreshThreadPtr refresh =
    new SessionRefreshThread(
        IceUtil::Time::seconds(
        router->getSessionTimeout()/2), session);
```

The next step is to alter the client configuration to set a default router (instead of a default locator):

```
# Client side Configuration
Ice.Default.Router=Mp3Ripper.Glacier2/router:tcp
-h 1.2.3.4 -p 10005
```

If we run this client (after deploying and starting the Glacier2 router), we get the following error:

```
$ ./client testcase.wav
Exception: Outgoing.cpp:368:
ObjectNotExistException:
object does not exist:
identity: `2DAD6FAB-200D-443F-83DB-C38AEF67C690'
facet:
operation: encode
```

The problem here is that Glacier2 is getting in the way. Glacier prevents clients from talking to objects they should not have access to and, unless told otherwise, Glacier2 will block *all* traffic. To coexist with Glacier2, at run time, IceGrid tells Glacier2 via the `Glacier2::SessionControl` interface to open appropriate holes in the firewall, so IceGrid clients can access the `IceGrid::Query` and `IceGrid::Session` objects (but no other objects). For allocated objects, IceGrid tells Glacier2 to additionally allow access to the allocated objects' identity. However, this means that Glacier2 blocks the client's attempt to contact the encoder object because the encoder object is not an allocated object (only the encoder *factory* is). The easiest way to fix this is to change the allocation method from object allocation to server allocation. With server allocation, IceGrid tells Glacier2 to allow access to *all* objects hosted by a server's indirect object adapters. Therefore, we can change the template as follows:

```
<icebox id="${instance-name}" exe="icebox"
 activation="on-demand" allocatable="true">
```

Or alternatively:

```
<icebox id="${instance-name}" exe="icebox"
 activation="session">
```

The `allocatable="true"` tag tells IceGrid to allocate the entire server to the client instead of just a single object. (`activation="session"` implies `allocation="true"`.) This also causes the IceGrid node to execute the server when the session allocates the server, and to shut the server down when the session releases it.

How do you decide whether to use session activation or straight server allocation? Session activation is appropriate if you want to run the server as a given user-ID, or if you want to use SSL with multiple client-side certificates. Otherwise, it is generally better to use server allocation.

SSL and Glacier2

Next we will add SSL to the mix, but we will still use user name and password authentication. This eliminates the weakness of the preceding approach, which sends the user name and password in clear text over a TCP connection (and hence is subject to eavesdropping attacks.) Here is a summary of the steps required to use SSL:

- Generate a certificate for the Glacier2 router.
- Configure the Glacier2 router to use only SSL for its client-side endpoints.
- Distribute the CA certificate to the client-side application.
- Modify the client-side default router proxy to use SSL.
- Configure the client-side SSL plug-in, and set the trust rules so that only the Glacier2 router is trusted.

For our initial configuration, we will authenticate the client with a user name and password, and secure communications with SSL.

First, we initialize the simple CA and generate a server certificate for the Glacier2 router. (We use the name `Glacier2 Router`, which results in the common name `Ice Server Glacier2 Router`.) We store the certificate in `glacier2_cert.pem`, and the private key in `glacier2_key.pem`. Then we change the Glacier2 router deployment to use SSL:

```
# IceGrid Deployment Descriptor
<server-instance template="Glacier2"
 client-endpoints="ssl -h 1.2.3.4 -p 10005"
 server-endpoints="tcp"
 session-timeout="30">
  <properties>
    <property name="Glacier2.SessionManager"
     value="EncoderIceGrid/SessionManager"/>
    <property name="Glacier2.PermissionsVerifier"
     value="EncoderIceGrid/NullPermissionsVerifie
r"/>
    <property name="Ice.Plugin.IceSSL"
     value="IceSSL:createIceSSL"/>
    <property name="IceSSL.CertAuthFile"
     value="ca_cert.pem"/>
    <property name="IceSSL.CertFile"
     value="glacier2_cert.pem"/>
    <property name="IceSSL.KeyFile"
     value="glacier2_key.pem"/>
  </properties>
</server-instance>
```

We also need to change the client configuration to use the CA and Glacier2's SSL endpoint:

```
# Client configuration
Ice.Default.Router=Mp3Ripper.Glacier2/router:ssl
-h 1.2.3.4 -p 10005
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
```

Note that since the client is anonymous (from the SSL point of view) it does not have a certificate itself. However, it must still identify the CA certificate so that the client knows how to validate that the router's certificate is actually valid. In other words, the client can be sure that it is talking to the correct Glacier2 instance, but Glacier2 cannot authenticate the client. When we try to run the client, we get the following error:

```
$ ./client  testcase1.wav
./client: ConnectorI.cpp:143: ProtocolException:
protocol exception:
SSL error for new outgoing connection:
remote address = 1.2.3.4:10005
sslv3 alert handshake failure: SSL alert number 40
```

The problem here is that the Glacier2 router tries to validate the client certificate but, because it has no such certificate, it rejects the connection. We can fix this by setting `IceSSL.VerifyPeer` property in the Glacier2 configuration to zero, which means that the client validates the server certificate, but the server does not request a certificate from the client. (To have Glacier2 recognize the property, we need to update its deployment and restart it.)

```
# IceGrid Deployment Descriptor
<property name="IceSSL.VerifyPeer" value="0"/>
```

Now the client works as expected. However, there is still something missing: as it stands, the client trusts any certificate signed by the CA certificate in `ca_cert.pem`. However, we want the client to trust only the Glacier2 router. To achieve this, we need to add a trust rule to the client configuration file:

```
# Client Configuration
IceSSL.TrustOnly=CN="Ice Server Glacier2 Router"
```

Now the client authenticates the Glacier2 router and validates that it indeed has the correct common name. The server authenticates the client via user name and password. Because communications are encrypted with SSL, this is now safe because the user name and password are no longer sent as clear text.

Secure Connection Authentication with Glacier2

Next, we will change the authentication method to use the credentials associated with the SSL connection instead of using a user name and password. This requires issuing a unique certificate to each client.

Here is a summary of the steps required to do this:

- Write an implementation of the `Glacier2::SSLPermissionsVerifier` interface. Because the server lives behind the firewall, it is not necessary to use SSL endpoints for it.
- Modify the Glacier2 router configuration to use the SSL permissions verifier. In addition, Glacier2 must use `IceSSL.VerifyPeer=2` (the default) because every client is now required to have a certificate.
- Modify the client's configuration to use a per-client certificate.
- Modify the client to authenticate via a secure connection instead of a user name and password.

As the client certificate, we can re-use the certificate from the SSL section. Here is the new Glacier2 deployment:

```
# IceGrid Deployment Descriptor
<server-instance template="Glacier2"
 client-endpoints="ssl -h 1.2.3.4 -p 10005"
 server-endpoints="tcp"
 session-timeout="30">
 <properties>
  <property name="Glacier2.SSLSessionManager"
   value="EncoderIceGrid/SessionManager"/>
  <property name="Glacier2.SSLPermissionsVerifier"
   value="SSLPermissionsVerifier"/>
  <property name="Ice.Plugin.IceSSL"
   value="IceSSL:createIceSSL"/>
  <property name="IceSSL.CertAuthFile"
   value="ca_cert.pem"/>
  <property name="IceSSL.CertFile"
   value="glacier2_cert.pem"/>
  <property name="IceSSL.KeyFile"
   value="glacier2_key.pem"/>
 </properties>
</server-instance>
```

Compared to the previous one, this deployment makes the following changes:

- It sets the property `Glacier2.SSLSessionManager` instead of `Glacier2.SessionManager`.
- It sets the property `Glacier2.SSLPermissionsVerifier` instead of `Glacier2.PermissionsVerifier`. (In this case, we have told the Glacier2 router to use the SSL permissions verifier that we developed earlier in this article. Note that we are also using TCP for the verifier since the verifier is behind the firewall.)
- It removes the setting of the `IceSSL.VerifyPeer` property because we now want to request a peer certificate.

Next, we need to alter the client configuration to supply a client-side certificate:

```
# Client Configuration
Ice.Default.Router=Mp3Ripper.Glacier2/router:ssl
-h 1.2.3.4 -p 10005
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
IceSSL.CertFile=matthew_cert.pem
IceSSL.KeyFile=matthew_key.pem
IceSSL.TrustOnly=CN="Ice Server Glacier2 Router"
```

The only other change to the client is to have it authorize via its secure connection as follows:

```
// C++
IceGrid::SessionPrx session;
try
{
    session = IceGrid::SessionPrx::uncheckedCast(
     router->createSessionFromSecureConnection());
}
catch(const IceGrid::PermissionDeniedException&ex)
{
    cout << "permission denied:\n" << ex.reason
        << endl;
    return 1;
}
```

Conclusion

You can protect unauthorized access to grid resources using Glacier2 only, SSL only, or a combination of both. Which mechanism to use is an administrative decision. If you deploy the grid inside a corporate network and do not want to allow access to the grid from the outside, the most likely choice is to secure administrative access to the grid with SSL, and to allow everything else to proceed unsecured. However, if you want to allow access to your grid from outside the boundaries of your corporate firewall, then you will certainly want to use Glacier2. Completely securing the grid with SSL is necessary only if you want strong security behind the firewall (that is, you assume that hostile parties may be present inside the network) and you do not want to set up a separate internal private network just to host the grid.

If you want to use Glacier2 to permit access to a grid from an outside network, I recommend the following configuration:

- Use SSL for communication between clients and Glacier2.
- Authenticate the Glacier2 router to clients with a certificate, so clients can be sure that they are talking to the correct Glacier2 server (instead of an impostor).
- Use user name and password client authentication with Glacier2.
- Run the Glacier2 router(s) on a server with two network cards, one facing the internal network and one facing the external network.
- Do not permit administrative access to Glacier2 or IceGrid from outside the firewall. For administrative access from inside the firewall, you can either leave admin access unsecured (if you trust everyone with access to the inside network), or you can secure admin access with a single SSL certificate. For slightly better protection, you can secure admin access with user name and password authentication, or with separate SSL certificates.
- Otherwise, run the grid on the internal network over TCP/IP. This provides better application performance (but assumes that the data your clients and servers exchange is either not sensitive or, if it is, that there are no eavesdroppers on the internal network).

This configuration is quite simple to set up and provides good security. In addition, because a Glacier2 router acts as a connection concentrator, if you have many clients, you can achieve better scalability by running several Glacier2 router instances.

In contrast, if you do not allow outside clients to access a grid through a firewall, you can dispense with Glacier2. In that case, you have the following options for running the grid:

1. Insecure: The grid is not protected in any way.
2. Administratively secure: The grid can be used by any client, but cannot be administered except by authenticated administrators.
3. Fully secure: Only authenticated and authorized clients and administrators can use the grid, with all communication explicitly secured via SSL.

For many organizations the second option is most attractive. Administration of the grid is secured (with SSL for encryption and user names and passwords for authentication), but any client on the internal network can use the grid without authentication. This is secure provided that the applications that run on the grid are secure, and it is simple to configure and manage. (If you want to use grid sessions with your application, you will need some form of client authentication; however, you can use a null permissions verifier if all you need are sessions, but no actual authentication.)

Of course, these are only general guidelines. If you have more complex requirements, we can provide assistance via our consulting services—please contact sales@zeroc.com if you are interested.

There are other aspects of grid security that I did not touch on in this article. For example, the way applications are deployed, the user-IDs of server processes, and the deployment of IceGrid nodes and the registry are also important aspects of grid security. I will discuss these and other security concerns in future articles on IceGrid.

FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about Ice. The questions and answers are taken from our support forum at http://www.zeroc.com/vbulletin/ and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

Q: Is Ice thread-safe?

The short answer is "yes". All Ice APIs are thread-safe, in the sense that you can call them concurrently from different threads without running the risk of corrupting data structures that are internal to the Ice run time. For example, it is perfectly safe to do the following:

```
// C++
// Executed by thread 1:
communicator->add(servant, id);

// ...
// Executed concurrently by thread 2:
communicator->remove(id);
```

The concurrent calls to `add` and `remove` are safe because the Ice run time internally locks the ASM before attempting to modify it, so the two API calls are serialized. In general, you never need to protect Ice-internal data structures from concurrent access.

However, Ice does not protect the integrity of application data. Consider the following Slice definition:

```
// Slice
struct Item
{
    string name;
    // ...
};

interface ItemStore
{
    void put(string s, Item i);
    bool get(string s, out Item i);
};
```

The `slice2cpp` compiler generates the following signatures for the `put` and `get` operations:

```
// C++
void put(const string& s, const Item& i);
bool get(const string& s, Item& i);
```

Now, if you call `put` and `get` concurrently and pass the same item to each operation, you can end up in trouble:

```
// C++
Item i;
i.name = "Joe";

// Pass i to thread 1 and thread 2...

// In thread 1:
itemStoreProxy->put("Joe", i);

// Concurrently in thread 2:
itemStoreProxy->get("Fred", i);
```

Of course, the problem here is that the `put` operation in thread 1 may read the contents of `i` at the same time as the `get` operation in thread 2 modifies the contents of `i`. The most likely outcome is that that the `put` operation is passed a corrupted value of `I` but, depending on your CPU architecture and threading package, the consequences might be more serious, such as a core dump (either while `get` and `put` are executing, or later, when the application attempts to use `i).`

So, as usual, you must establish critical regions around concurrent access to application data, regardless of whether that access is performed by a thread of your own or a thread inside the Ice run time.

Note that "thread-safe" does not mean that you can blindly call any Ice API at any time. There are a few Ice API calls that can cause deadlock. For example:

```
// Slice
interface Admin
{
    void shutdown(); // Shut down server
};
```

```
// C++
void
AdminI::shutdown(const Current& c)
{
    c.adapter->deactivate();
    // ...
    c.adapter->waitForDeactivate(); // Deadlock!
}
```

This cannot possibly work because `waitForDeactivate` waits until all operations on the adapter have completed; if you call `waitForDeactivate` from within an operation on the adapter being deactivated, `waitForDeactivate` cannot complete until the operation has completed, and the operation cannot complete until `waitForDeactivate` has completed, so the code deadlocks. However, there are only a handful of operations that can cause this problem: `Communicator::destroy`, `Communicator::waitForShutdown`, `Adapter::waitForDeactivate`, `Adapter::waitForHold`, and `Service::waitForShutdown`. If your code deadlocks due to incorrect use of these operations, the problem is easily diagnosed: one of the threads will be stuck in one of these methods and have an earlier stack frame that corresponds to the implementation of a Slice operation.

Q: Why does the MFC leak detector report memory leaks with Ice?

When running a debug version of an MFC application that links with the Ice libraries, after the application terminates, you may see something similar to the following in the IDE debug window:

```
Detected memory leaks!
Dumping objects ->
{77} normal block at 0x00475768, 16 bytes long.
Data: < WG hWG > 18 57 47 00 68 57 47 00 00 00 00
00 00 00 00 00
{75} normal block at 0x00475A90, 1808 bytes long.
Data: <0123456789ABCDEF> 30 31 32 33 34 35 36 37
38 39 41 42 43 44 45 46
{74} normal block at 0x00472FE0, 28 bytes long
...
```

We have tested the Ice run time extensively and are confident that it does not harbor memory leaks—even when MFC is used. The erroneous leak reports are caused by the use of static variables in the Ice libraries, such as instances of `std::string`. These instances allocate memory when they are constructed, but have not released the memory at the time the Microsoft memory tracker runs its leak detection routines. This is annoying and, unfortunately, there is no known solution to this problem.

Of course, it is always possible that your application has real memory leaks, so it is best to use a leak detector to help track down such bugs. However, we recommend that you ignore the output from the Microsoft memory tracker and instead use a leak detector that does not suffer from this problem, such as Rational Purify.