Securing a Glacier2 Router

As a firewall, a Glacier2 router represents a doorway into a private network, and in most cases that doorway should have a good lock. The obvious first step is to use SSL for the router's client endpoints. This allows you to secure the message traffic and restrict access to clients having the proper credentials. However, the router takes security even further by providing access control and filtering capabilities.

On this page:

Glacier2 Access Control

The authentication capabilities of SSL may not be sufficient for all applications: the certificate validation phase of the SSL handshake verifies that the user is who he says he is, but how do we know that he should be allowed to use the router? Glacier2 addresses this issue through the use of an access control facility that supports two forms of authentication: passwords and certificates. You can configure the router to use whichever authentication method is most appropriate for your application, or you can configure both methods in the same router.

Password Authentication

The router verifies the user name and password arguments to its `createSession` operation before it forwards any requests on behalf of the client. Given that the password is sent "in the clear," it is important to protect these values by using an SSL connection with the router.

There are two ways for the router to verify a user name and password. By default, the router uses a file-based access control list, but you can override this behavior by installing a proxy for an application-defined verifier object. Configuration properties define the password file name or the verifier proxy; if you install a verifier proxy, the password file is ignored. Since we have already discussed the password file, we will focus on the custom verifier interface here.

An application that has special requirements can implement the interface `Glacier2::PermissionsVerifier` to gain programmatic control over access to a router. This can be especially useful in situations where a repository of account information already exists (such as an LDAP directory), in which case duplicating that information in another file would be tedious and error-prone.

The Slice definition for the interface contains just one operation:

Slice

```
module Glacier2 {
    interface PermissionsVerifier {
        idempotent bool checkPermissions(string userId, string password, out string reason)
            throws PermissionDeniedException;
    };
};
```

The router invokes `checkPermissions` on the verifier object, passing it the user name and password arguments that were given to `createSessior`. The operation must return true if the arguments are valid, and false otherwise. If the operation returns false, a reason can be provided in the output parameter. Starting with Ice 3.5, `checkPermissions` can also throw `Glacier2::PermissionDeniedException`. Glacier2 forwards this exception as-is to the client, which means the verifier can raise a subclass of `PermissionDeniedException` in order to provide more information to the client.

To configure a router with a custom verifier, set the configuration property `Glacier2.PermissionsVerifier` with the proxy for the object.

In situations where authentication is not necessary, such as during development or when running in a trusted environment, you can use Glacier2's built-in "null" permissions verifier. This object accepts any combination of user name and password, and you can enable it with the following property definition:

```
Glacier2.PermissionsVerifier=Glacier2/NullPermissionsVerifier
```

Note that the category of the object's identity (`Glacier2` in this example) must match the value of the property `Glacier2.InstanceName`.

Example

A sample implementation of the `PermissionsVerifier` interface is provided in the `demo/Glacier2/callback` directory.

Certificate Authentication

The `createSessionFromSecureConnection` operation does not require a user name or password because the client's SSL connection to the router already supplies the credentials necessary to sufficiently identify the client, in the form of X.509 certificates.

It is up to you to decide what constitutes sufficient identification. For example, a single certificate could be shared by all clients if there is no need to distinguish between them, or you could generate a unique certificate for each client or a group of clients. Glacier2 does not enforce any particular policy, but simply delegates the decision of whether to accept the client's credentials to an application-defined object that implements the `Glacier2::SSLPermissionsVerifier` interface:

Slice

```
module Glacier2 {
    interface SSLPermissionsVerifier {
        idempotent bool authorize(SSLInfo info, out string reason)
            throws PermissionDeniedException;
    };
};
```

Router clients may only use `createSessionFromSecureConnection` if the router is configured with a proxy for an `SSLPermissionsVerifier` object. The implementation of `authorize` must return true to allow the client to establish a session. To reject the session, `authorize` must return false and may optionally provide a value for `reason`, which is returned to the client as a member of `PermissionDeniedException`. Starting with Ice 3.5, it can also throw `Glacier2::PermissionDeniedException`. Glacier2 forwards this exception as-is to the client, which means the verifier can raise a subclass of `PermissionDeniedException` in order to provide more information to the client.

The verifier examines the members of `SSLInfo` to authenticate a client:

Slice

```
module Glacier2 {
    struct SSLInfo {
        string remoteHost;
        int remotePort;
        string localHost;
        int localPort;
        string cipher;
        Ice::StringSeq certs;
    };
};
```

The structure includes address information about the remote and local hosts, and a string that describes the ciphersuite negotiated for the SSL connection between the client and the router. These values are generally of interest for logging purposes, whereas the `certs` member supplies the information the verifier needs to make its decision. The client's certificate chain is represented as a sequence of strings that use the Privacy Enhanced Mail (PEM) encoding.

The first element of the sequence corresponds to the client's certificate, followed by its signing certificates. The certificate of the root Certificate Authority (CA) is the last element of the sequence. An empty sequence indicates that the client did not supply a certificate chain.

Although the certificate chain has already been validated by the SSL implementation, a verifier implementation typically needs to examine it in detail before making its decision. As a result, the verifier will need to convert the contents of `certs` into a more usable form. Some Ice platforms, such as Java and .NET, already provide certificate abstractions, and IceSSL supplies its own for C++ users. IceSSL for Java and .NET defines the method `IceSSL.Util.createCertificate`, which accepts a PEM-encoded string and returns an instance of the platform's certificate class. In C++, the class `IceSSL::Certificate` has a constructor that accepts a PEM-encoded string.

In addition to examining certificate attributes such as the distinguished name of the subject and issuer, it is also important that a verifier consider the length of the certificate chain.

To install your verifier, set the `Glacier2.SSLPermissionsVerifier` property with the proxy of your verifier object.

In situations where authentication is not necessary, such as during development or when running in a trusted environment, you can use Glacier2's built-in "null" permissions verifier. This object accepts the credentials of any client, and you can enable it with the following property definition:

```
Glacier2.SSLPermissionsVerifier=Glacier2/NullSSLPermissionsVerifier
```

Note that the category of the object's identity (`Glacier2` in this example) must match the value of the property `Glacier2.InstanceName`.

Interaction with a Permissions Verifier

The router attempts to contact the configured permissions verifiers at startup. If an object is unreachable, the router logs a warning message but continues its normal operation (you can suppress the warning using the `--nowarn` option.) The router does not contact a verifier again until it needs to invoke an operation on the object. For example, when a client asks the router to create a new session, the router makes another attempt to contact the verifier; if the object is still unavailable, the router logs a message and returns `PermissionDeniedException` to the client.

Obtaining SSL Credentials for a Router Client

Servers that wish to receive information about a client's SSL connection to the router can define the `Glacier2.AddConnectionContext` property. When enabled, the router adds several entries to the request context of each invocation it forwards to a server, providing information such as the client's encoded certificate (if supplied) and addressing details. If the client's connection uses SSL, the router defines the `_con.peerCert` entry in the context. A server can check for the presence of this entry and also extract additional context entries as shown below in this example:

C++

```
void unlockDoor(string id, const Ice::Current& curr)
{
    Ice::Context::const_iterator i = curr.ctx.find("_con.peerCert");
    if (i != curr.ctx.end()) {
        string certPEM;
        certPEM = i->second;
        cout << "Client address = "
             << curr.ctx["_con.remoteAddress"]
             << ":" << curr.ctx["_con.remotePort"] << endl;
        ...
    }
    ...
}
```

If the client supplied a certificate, the server can decode and examine it using the techniques discussed for IceSSL.

Request Filtering

The Glacier2 router is capable of filtering requests based on a variety of criteria, which helps to ensure that clients do not gain access to unintended objects.

Address Filters

To prevent a client from accessing arbitrary back-end hosts or ports, you can configure a Glacier2 router to validate the address information in each proxy that the client attempts to use. Two properties determine the router's filtering behavior:

- `Glacier2.Filter.Address.Accept`
  An address is accepted if it matches an entry in this property and does not match an entry in `Glacier2.Filter.Address.Reject`.

- `Glacier2.Filter.Address.Reject`
  An address is rejected if it matches an entry in this property.

The value of each property is a list of *address*:*port* pairs separated by spaces, as shown in the example below:

```
Glacier2.Filter.Address.Accept=192.168.1.5:4063 192.168.1.6:4063
```

This configuration allows clients to use only two hosts in the back-end network, and only one port on each host. A client that attempts to use a proxy containing any other host or port receives an `ObjectNotExistException` on its initial request.

You can also use ranges, groups and wildcards when defining your address filters. For example, the following property value shows how to use an address range:

```
Glacier2.Filter.Address.Accept=192.168.1.[5-6]:4063
```

This property is equivalent to the first example, but the range notation allows us to define the filter more concisely. Similarly, we can restate the property using the group notation by separating values with a comma:

```
Glacier2.Filter.Address.Accept=192.168.1.[5,6]:4063
```

The wildcard notation uses the * character to substitute for a value:

```
Glacier2.Filter.Address.Accept=10.0.*.1:4063
```

The range, group, and wildcard notation is also supported when specifying ports, as shown below:

```
Glacier2.Filter.Address.Accept=192.168.1.[5,6]:[10000-11000]
Glacier2.Filter.Address.Reject=192.168.1.[5,6]:[10500,10501]
```

In this configuration, the router allows clients to access all of the ports in the range `10000` to `11000`, except for the two ports `10500` and `10501`.

At first glance, you might think that the following property definition is pointless because it would prevent clients from accessing any back-end server:

```
Glacier2.Filter.Address.Reject=*
```

In reality, this configuration only prevents clients from accessing servers using direct proxies, that is, proxies that contain endpoints. As a result, the property causes Glacier2 to accept only indirect proxies.

By default, a Glacier2 router forwards requests for any address, which is equivalent to defining the property `Glacier2.Filter.Address.Accept=*`.

Category Filters

The `Ice::Identity` type contains two string members: category and name. You can configure a router with a list of valid identity categories, in which case it only routes requests for objects in those categories. The configuration property `Glacier2.Filter.Category.Accept` supplies the category list:

```
Glacier2.Filter.Category.Accept=cat1 cat2
```

This property does not affect the routing of callback requests from back-end servers to router clients.

By default a Glacier2 router forwards requests for any category.

If a category contains spaces, you can enclose the value in single or double quotes. If a category contains a quote character, it must be escaped with a leading backslash.

Glacier2 can optionally manipulate the category filter automatically. When you set `Glacier2.Filter.Category.AcceptUser` to a value of 1, the router adds the session's user name (for password authentication) or distinguished name (for SSL authentication) to the list of accepted categories. To ensure the uniqueness of your categories, you may prefer setting the property to a value of 2, which causes the router to prepend an underscore to the user name or distinguished name before adding it to the list.

A session manager can also configure category filters dynamically using Glacier2's `SessionControl` interface.

Identity Filters

The ability to filter on identity categories, as described in the previous section, is a convenient way to limit clients to particular groups of objects. For even stricter control over the identities that clients are allowed to access, you can use the `Glacier2.Filter.Identity.Accept` property. The value of this property is a list of identities, separated by whitespace, representing the *only* objects the router's clients may use.

If an identity contains spaces, you can enclose the value in single or double quotes. If an identity contains a quote character, it must be escaped with a leading backslash.

Clearly, specifying a static list of identities is only practical for a small set of objects. Furthermore, in many applications, the complete set of identities cannot be known in advance, such as when objects are created on a per-session basis and use UUIDs in their identities. For these situations, category-based filtering is generally sufficient. However, a session manager can also use Glacier2's dynamic filtering interface, `SessionControl`, to manage the set of valid identities at run time.

Adapter Filters

Applications often use IceGrid in their back-end network to simplify server administration and take advantage of the benefits offered by indirect proxies. Once you have configured Glacier2 with an appropriate locator proxy, clients can use indirect proxies to refer to objects in IceGrid-managed servers. Indirect proxies come in two forms: one that contains only an identity, and one that contains an identity and an object adapter identifier. You can use the category and identity filters described in previous sections to control identity-only proxies, and you can use the property `Glacier2.Filter.AdapterId.Accept` to enforce restrictions on indirect proxies that use an object adapter identifier.

For example, the following property definition allows a client to use the proxy `factory@WidgetAdapter` but not the proxy `factory@SecretAdapter`:

```
Glacier2.Filter.AdapterId.Accept=WidgetAdapter
```

If an adapter identifier contains spaces, you can enclose the value in single or double quotes. If an adapter identifier contains a quote character, it must be escaped with a leading backslash.

A session manager can also configure this filter dynamically using Glacier2's `SessionControl` interface.

Proxy Filters

The Glacier2 router maintains an internal routing table that contains an entry for each proxy used by a router client; the size of the routing table grows in proportion to the number of clients and their proxy usage. Furthermore, the amount of memory that the routing table consumes is affected by the number of endpoints in each proxy. Glacier2 provides two properties that you can use to limit the size of the routing table and defend against malicious router clients.

The property `Glacier2.RoutingTable.MaxSize` specifies the maximum number of entries allowed in the routing table. If the size of the table exceeds the value of this property, the router evicts older entries on a least-recently-used basis. (Eviction of proxies from the routing table is transparent to router clients.) The default size of the routing table is 1000, but you may need to define a different value depending on the needs of your application. While experimenting with different values, you may find it useful to define the property `Glacier2.Trace.RoutingTable` to see a log of the router's activities with respect to the routing table.

The property `Glacier2.Filter.ProxySizeMax` sets a limit on the size of a stringified proxy. The Ice run time places no limits on the size of proxy components such as identities and host names, but a malicious client could manufacture very large proxies in a denial-of-service attack on a Glacier2 router. By setting this property to a reasonably small value, you can prevent proxies from consuming excessive memory in the router process.

Client Impact

The Glacier2 router immediately terminates a client's session if it attempts to use a proxy that is rejected by an address filter or exceeds the size limit defined by the property `Glacier2.Filter.ProxySizeMax`. The Ice run time in the client responds by raising `ConnectionLostException` to the application.

For category, identity, and adapter identifier filters, the router raises `ObjectNotExistException` if any of the filters rejects a proxy and none of the filters accepts it.

To obtain more information on the router's reasons for terminating a session or rejecting a request, set the following property and examine the router's log output:

```
Glacier2.Client.Trace.Reject=1
```

Glacier2 Routing Table

The Glacier2 router maintains an internal routing table for each session. The routing table holds every proxy used by its session. Consequently, the size of the routing table grows in proportion to the number of proxies used by a session. Furthermore, the amount of memory that all of the routing tables consume grows with the number of active sessions.

The property `Glacier2.RoutingTable.MaxSize` allows you to specify an upper limit on the number of entries in the routing table. If the size of the table exceeds the value of this property, the router evicts older entries on a least-recently-used basis. (Eviction of proxies from the routing table is transparent to router clients.) The default size of the routing table is 1000, but you may need to define a different value depending on the needs of your application. While experimenting with different values, you may find it useful to define the property `Glacier2.Trace.RoutingTable` to see a log of the router's activities with respect to the routing table.

The router does not remove entries from a session's routing table except when evicting an old entry to make room for a new one. In particular, an exception that occurs while routing a request for a proxy does *not* cause that proxy to be removed from the routing table. Note however that the routing table is destroyed upon session destruction.

Glacier2 Administrative Interface

Glacier2 supports an administrative interface that allows you to shut down a router programmatically:

Slice

```
module Glacier2 {
    interface Admin {
        idempotent void shutdown();
    };
};
```

To prevent unwanted clients from using the `Admin` interface, the object is only accessible on the endpoints defined by the `Glacier2.Admin.Endpoints` property. This property has no default value, meaning you must define the property in order to make the `Admin` object accessible.

If you decide to define `Glacier2.Admin.Endpoints`, choose your endpoints carefully. We generally recommend the use of endpoints that are accessible only from behind a firewall.

See Also

- Glacier2 Properties
- IceSSL
- Getting Started with Glacier2
- Callbacks through Glacier2
- Dynamic Request Filtering with Glacier2
- IceGrid and Glacier2 Integration