Inter-ORB Message Protection in the ILU system

Bill Janssen
Xerox Palo Alto Research Center
12 September, 1995

ILU is a distributed object system implemented at Xerox's Palo Alto Research Center that has many of the attributes of a CORBA 'library' ORB; that is, an ORB implemented as a library which is linked with each application that wishes to use it. This document describes part of the ORB security system implemented for the 1.9 release of ILU (which should be available 4Q95), and designed at PARC during 1995 by (in alphabetical order) Antony Courtney, Bill Janssen, and Mike Spreitzer. More information on ILU, and the sources to the ILU libraries and stubbers, are available via FTP from PARC. See the document ftp://ftp.parc.xerox.com/pub/ilu/ilu.html for more details.

The ILU security system is implemented in a number of pieces at a number of different layers. It is designed to be useful for a variety of authorization, identity, accounting, and auditing schemes, but those mechanisms are not the topic of this document, which addresses mainly the provisions for inter-ORB message protection. Each instance of an ILU ORB is defined to operate in a single address space, but ILU provides for transparent communication between ILU ORBs when an operation is invoked on a foreign (outside the calling ORB) object reference. ILU provides a generic scheme for securing inter-ORB communication, using the IETF Common Authentication Technology working group's Generic Security Service (GSS) API. (The ILU distribution also provides an implementation of the GSS, which contains a meta-object protocol allowing specific security mechanisms to be registered with and used by the GSS shell implementation.) Inside the generic mechanism, specific security mechanisms can be selected to give various degrees of protection.

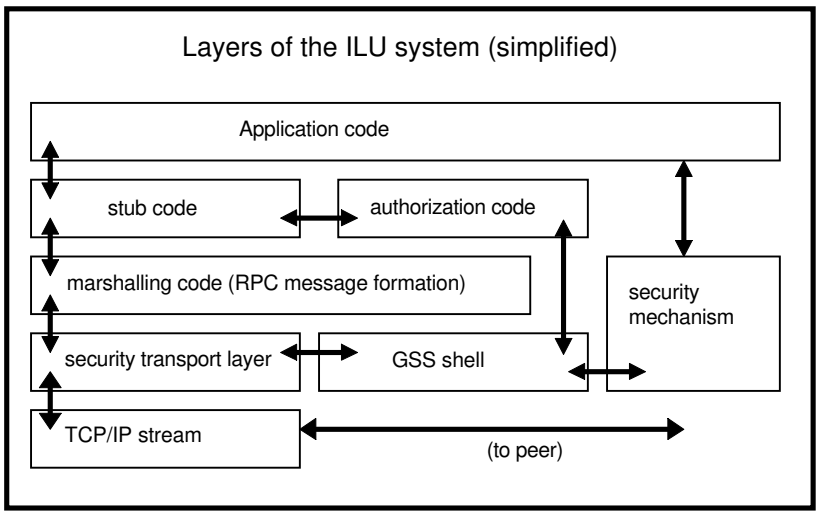
The GSS-based inter-ORB security provides message integrity and/or message privacy services, depending on which specific services are requested of the specific security mechanism selected for use. All parts of the message are protected; all messages on a secured transport are sent as record-marked GSS tokens. Multiple tokens may be sent for a single RPC-level message; some security mechanisms may insert empty padding messages to further confuse traffic analysis attacks. This inter-ORB security may also be used for ILU 'transport-buffer-based' object persistence mechanisms, but it is not clear to us what security implications this has.

Creating secure access channels to an Object Adapter

ILU secures communication channels independently of which particular RPC protocol or messaging structure is used for communication of inter-ORB messages. The ILU transport stack consists of the following pieces: an application layer, which calls into a caller-side stub layer, which calls into RPC-mechanism-specific marshalling code, which calls into one or more transport filters, arranged in a stack. For example, an application in

Copyright © 1995 Xerox Corporation

C++ may call into the stubs for some specific interface, which call down into the IIOP message marshalling code, which calls down to the compression transport layer (passing it either whole IIOP messages or fragments thereof), which calls down to the security transport layer (which turns byte streams passed from above into security data tokens), which calls down to a simple record-marking layer (which puts 32-bit length headers on each GSS data token), which calls down to the TCP/IP transport layer (which actually writes the bytes to the TCP/IP stream). Thus, each message passed between ORBs in this example is a record-marked GSS token containing an encrypted signed compressed fragment of a full IIOP message. The fragment may be a 'improper' fragment -- a whole IIOP message. Note that this is only one possible ILU transport stack; the layers may be arranged in other orders, within some constraints. In particular, our current design requires the security transport layer to be immediately above a GSS token record-marking layer, which in turn must be immediately above the actual transport layer (the layer which actually writes to the TCP/IP buffers, for instance). The GSS token record-marking layer may adapt to the particular security scheme being used; in particular, if the wire format of the security mechanism already specifies record-marking information, the record-marking layer will simply interpret that information, rather than adding any new information to the packets passing through it.



ILU objects are managed by Object Adapters, called in ILU 'kernel servers'. There may be multiple OA's per ORB. Each OA 'owns' some set of ILU objects instances, and manages certain functionality for those instances. One item of functionality managed by an OA is communication with OA's in other ORBs. This inter-OA communication is always performed between a 'true' OA and its counterpart in another ORB, the 'surrogate' OA. Instances in a true OA are those which actually contain implementations for the operations of the instances' types, so the true OA may be thought of as the 'server-side' OA. A surrogate OA manages surrogate or proxy instances for some subset of the instances managed by its corresponding true OA, so the surrogate OA may be thought of as the 'client-side' OA. There may be many surrogate OAs corresponding to any single

Copyright © 1995 Xerox Corporation

true OA, but there may only be one surrogate OA for each true OA in any one ORB. All inter-ORB communication is between a surrogate OA and its corresponding true OA.

Selection of security mechanisms for access to instance operations is done by calls to true OA's, establishing server-side 'ports'. Each port describes a communication mechanism which surrogate OA's may use in communicating with the true OA. Each port may independently select an RPC protocol, a transport stack, and (if the transport stack includes the security transport layer) identify which security mechanism to use, and the server-side identity to be used in establishing security contexts via that security mechanism. Communication with that true OA by any of its corresponding surrogate OA's may then be accomplished through any of the ports on the OA.

When a client decides to use a particular object, it consults the IOR of the object to discover communication parameters for that object, including the parameters of the various ports of the true OA for the object. The client then creates a surrogate OA for the true OA, if one does not already exist in its ORB, and indicates which types of communication channels it wishes to use. A communication channel is an abstraction which incorporates the notions of RPC protocol (say, GIOP, though others are possible with ILU), message transport (say, TCP/IP), compression (say, lz77), and security mechanism (say, Netscape's Secure Sockets Layer). Note that true OA's can force a particular communication channel type by either only opening one port, or only telling this particular client about one of its ports.

When a client opens a communications channel of a particular type, the surrogate OA initialization code automatically creates the proper RPC protocol object and transport layer stack on the client side, initializing each layer in the transport stack as appropriate. If the transport stack includes the security transport layer, it also initializes this layer. The security transport layer is written against the GSS C API, using no extensions. The initialization code retrieves the client's identity (from the per-thread context of the client), the server's identity (one of the parameters of the port, and passed as a parameter to the secure transport layer's initialization code), and the specific security mechanism to use (specified as a string, which may be either the GSS-specified string form of the security mechanism OID, or an ILU nickname for it (an optional feature)). The init code then calls gss_acquire_cred() with the client identity and security mechanism, and if that succeeds, calls gss_init_sec_context() with the client credentials, the server identity, and the security mechanism. Any context token output by that call is passed through the lower-level transport layers to the true OA, where it is passed to gss_accept_sec_context(). Any return token is passed back, etc, until the security context is established. This sequence of messages is called the security context handshake.

Note that at this time, and in all future uses of the channel, the actual transport mechanism, TCP/IP, is used only to convey GSS tokens back and forth, typically with some simple record-marking protocol which may either be provided by the particular wire format of the security mechanism (if any), or via an additional optional record marking

Copyright © 1995 Xerox Corporation

layer between the security layer and the TCP layer. No recognizable RPC messages are ever sent on the TCP connection.

Using the secure channel

When a call is made, the client calls down into the marshalling stubs with an indication of which channel is to be used for the call. If no port is specified, the OA makes the choice according to simple defaulting rules. Typically, a surrogate OA has only one channel open to its true OA, and the client accepts it by default. The marshalling stubs build up the appropriate RPC message, and dispatch it, either whole or in fragments, to the top layer of the transport stack. For the purpose of this discussion we'll use the term fragment, but it is important to remember that a whole message is also a valid 'fragment'. Each layer of the transport stack transforms the fragment in some way (which may be purely conceptual), and passes it to the next lower layer. In particular, the security transport layer takes the fragment and applies gss_wrap() to it, using the security context that was negotiated when the channel was set up. The wrap procedure may perform either message integrity on the fragment, or message privacy, or both. This produces a GSS data token, which is passed to the next lowest layer for possible additional transformation and eventual transport to the true OA. On the true side, the stack is reversed; a GSS data token coming in is passed to gss_unwrap(), which either verifies the MIC, or decrypts the token to produce a fragment, or both, and sends it up to the next highest level. (When we say 'up', it's important to note that ILU uses a pull model at this stage, rather than a push model, so really the data is being pulled up by some higher level.)

Note that all data tokens flowing across a particular TCP/IP connection in this scheme are using the same security context. It would be possible to support multiple security contexts on the same TCP/IP connection by prefixing each token with both a context ID in addition to the optional record-marking length. One motivation to not do such multiplexing would be a desire to stay wire-compatible with any security mechanisms which specific a wire format (such as Netscape's SSL). On the other hand, consistency of message format may be deemed more important than conformance to issues which security mechanism specs perhaps should not specify at all. A simple header with a token length, one bit to indicate whether the token is a context token or a data token (upcoming changes to the GSS may eliminate the need for this bit), and a context ID should be sufficient for most uses.

Operations outside the scope of the GSS

The GSS API only provides interfaces for part of the task of a typical security mechanism. In particular, each security mechanism must provide mechanisms to provide credentials for various identities in some mechanism-specific fashion, so that gss_acquire_cred() may succeed. In the ILU implementation of the GSS, an application is expected to communicate directly with the security mechanism via an API provided by the security mechanism for such operations as establishing credentials for a given identity. Security mechanisms may of course attempt to establish a default identity by inspection of the process in which it executes, but such mechanisms are deprecated. We have not

Copyright © 1995 Xerox Corporation

attempted to standardize an API for performing these operations, but look forward to the CAPI work with great interest.

Additionally, there are matters such as establishing a public key for a desired service identity on the client side. Our experience is limited in this area, but we feel that this is again a security-mechanism-specific operation, and leave the specific required API for performing such operations up to the implementation of the security mechanism.

Authorization tokens

ILU interposes stub code between client-side application code and ILU marshalling code, and between the server-side implementation of an operation and the ILU marshalling code. Authorization, auditing, and accounting services are invoked by this stub code both on the server side, to validate caller access, and on the client side (for replies), to validate server authority. Each of these mechanisms is passed two pieces of information, called 'authorization tokens', one derived from the security context of the channel over which a message is received, and the other from information passed in the message itself.

The first piece is called the 'channel client identity' (on the server side), which consists of the name of an identity, plus the OID of a security mechanism by which that identity has been authenticated. This identity is the one used by the client during the security context handshake. The identity may be null, in the case of a security mechanism which allows context establishment with anonymous clients. The second piece of authorization information is called the 'per-call authorization token'. This token is provided by the client-side application or some framework specific to the application domain, rather than by any ILU mechanism, and can be used for digital cash payments, certificates of authorization, or simple identities which are implicitly vouched for by the channel client identity. In the IIOP protocol, the per-call authorization token is carried in the Principal slot of the IIOP Request message header.

Some authorization mechanisms will prefer to use the channel client identity solely, others will prefer to ignore the channel client identity, and use application-specific per-call authorization tokens solely. Since many of the users of ILU wish to experiment with various techniques for doing authorization and accounting, the ILU system attempts to put minimal constraints on their freedom to perform those experiments.

We feel that a similar two-token scheme (channel server identity and per-reply authorization token) is desirable for replies. One server, with a particular identity, may be simply a trading house or dispatcher for services provided by other identities, so it seems necessary to have a per-call identity on replies, in addition to having a per-channel identity. Reply messages in various RPC protocols should be modified, if necessary, to allow this information to be transmitted.

Copyright © 1995 Xerox Corporation

—A—

authorization tokens, 5

—C—

CAPI, 5
channel client identity, 5
channel server identity, 5
communication channel, 3

—D—

digital cash, 5

—G—

GIOP, 3
GSS, 1
GSS C API, 3
gss_accept_sec_context()., 3
gss_acquire_cred(), 3
gss_init_sec_context(), 3
gss_unwrap(), 4
gss_wrap(), 4

—I—

ILU, 1
improper fragment, 2
IOR, 3

—K—

kernel servers, 2

—L—

library ORB, 1
lz77, 3

—M—

MIC, 4
multiple security contexts, 4

—P—

per-call authorization token, 5
per-reply authorization token, 5
port, 3
Principal, 5

—S—

Secure Sockets Layer, 3, 4
security context handshake, 3
surrogate OA, 2

—T—

TCP/IP, 3
traffic analysis attacks, 1
true OA, 2

Copyright © 1995 Xerox Corporation