GSS-API Programming Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–1331–10
May 2002

020115@3062

Contents

Preface

The *GSS-API Programming Guide* explains the Generic Security Services Application Programming Interface — the GSS-API. The GSS-API is a framework that allows developers to write applications that take advantage of security mechanisms such as Kerberos v5, without having to explicitly program for any one mechanism. Programs using the GSS-API therefore can be highly portable, not only from one platform to another, but from one security setup to another and from one transport protocol to another. The GSS-API provides several levels of data protection, consistent with the underlying security mechanisms that have been implemented on a system.

Who Should Use This Book

The *GSS-API Programming Guide* is intended for C-language developers who want to write programs that transfer data from one application to another securely, such as client-server programs. No specific knowledge of transport protocols or network programming is necessary to understand or use the GSS-API. (Of course, you will need to understand these areas in order to write networking applications, since the GSS-API does not itself perform transport.)

Before You Read This Book

You should be familiar with C programming. A basic knowledge of security mechanisms is helpful but not required. You do not need to have specialized knowledge about network programming to use this book.

How This Book Is Organized

Chapter 1 provides an overview of the GSS-API. It explains the general steps involved in using the GSS-API, covers the basic concepts, and details a few of the most important functions.

Chapter 2 is a walk-through of the sample programs listed in Appendix A.

Appendix A is a program listing for two sample programs: a GSS-API client and a GSS-API server.

Appendix B provides reference information on GSS-API functions, status codes, and data types.

Appendix C is a short discussion about specifying a security mechanism in the GSS-API.

Appendix D explains some features that are unique to Sun's implementation of the GSS-API.

Appendix E contains tables showing the status codes returned by the Kerberos v5 security mechanism.

Glossary is a list of words and phrases found in this book and their definitions.

Related Documentation

You might find the following to be helpful:

- *ONC+ Developer's Guide*

Two documents provide descriptions of the GSS-API (and are somewhat more oriented toward the GSS-API implementor than to the application developer). The Generic Security Service Application Program Interface document (`ftp://ftp.isi.edu/in-notes/rfc2743.txt`) provides a conceptual overview of the GSS-API, while the Generic Security Service API Version 2: C-Bindings document (`ftp://ftp.isi.edu/in-notes/rfc2744.txt`) discusses the specifics of the C-language-based GSS-API.

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

The GSS-API: An Overview

The Generic Security Standard Application Programming Interface (GSS-API) provides a way for applications to protect data that is sent to peer applications; typically, this might be from a client on one machine to a server on another. This chapter provides information on the following subjects:

Introduction to the GSS-API

As its name implies, the GSS-API enables programmers to write applications that are generic with respect to security; that is, they do not have to tailor their security implementations to any particular platform, security mechanism, type of protection, or transport protocol. Although the GSS-API *enables* applications control over security aspects, a programmer using GSS-API can write a program that is ignorant of the details of protecting network data. Therefore, a program that takes advantage of GSS-API is more portable as regards network security. More than anything else, this portability is the hallmark of the Generic Security Standard API.

The GSS-API does not actually provide security services itself. Rather, it is a framework that provides security services to callers in a generic fashion, supportable with a range of underlying mechanisms and technologies such as Kerberos v5 or public key technologies, as shown in Figure 1–1:

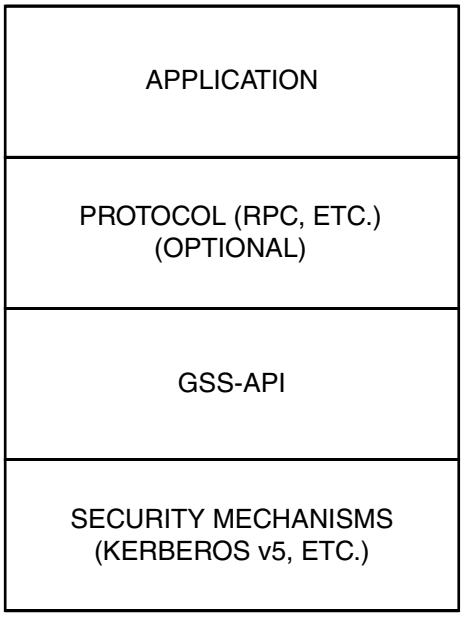

FIGURE 1–1 The GSS-API Layer

Broadly speaking, the GSS-API does two main things:

1. It creates a security *context* in which data can be passed between applications. A context can be thought of as a sort of "state of trust" between two applications. Applications that share a context know who each other are and thus can permit data transfers between them as long as the context lasts.
2. It applies one or more types of protection, known as *security services*, to the data to be transmitted. Security services are explained in "Security Services" on page 13.

Of course, the GSS-API is more complex than that. Some of the other things that the GSS-API does include: data conversion; error checking; delegation of user privileges; information display; and identity comparison. The GSS-API includes numerous support or convenience functions.

Application Portability

As mentioned above, the GSS-API provides several types of portability for applications:

- *Mechanism independence*. GSS-API provides a generic interface to the mechanisms for which it has been implemented. By specifying a default security mechanism, an application does not need to know which mechanism it is using (for example, Kerberos v5), or even what *type* of mechanism it uses. As an example, when an application forwards a user's credential to a server, it does not need to know if that credential has a Kerberos format or the format used by some other mechanism, nor how the credentials are stored by the mechanism and accessed by the application. (If necessary, an application can specify a particular mechanism to use.)
- *Protocol independence*. The GSS–API is independent of any communications protocol or protocol suite. It can be used with applications that use, for example, sockets, RCP, or TCP/IP.

  RPCSEC_GSS is an additional layer that smoothly integrates GSS-API with RPC. For more information, see "RPCSEC_GSS Layer" on page 14.
- *Platform independence*. The GSS-API is completely oblivious to the type of operating system on which an application is running.
- *Quality of Protection independence*. Quality of Protection (QOP) is the name given to the type of algorithm used in encrypting data or generating cryptographic tags; the GSS-API allows a programmer to ignore QOP, using a default provided by the GSS-API. (On the other hand, an application can specify the QOP if necessary.)

Security Services

The basic security offered by the GSS-API is *authentication*. Authentication is the verification of an identity: if you are authenticated, it means that you are recognized to be who you say you are.

The GSS-API provides for two additional security services, if supported by the underlying mechanisms:

- *Integrity*. It's not always sufficient to know that an application sending you data is who it claims to be. The data itself could have become corrupted or compromised. The GSS-API provides for data to be accompanied by a cryptographic tag, known as an Message Integrity Code (MIC), to prove that the data that arrives at your doorstep is the same as the data that the sender transmitted. This verification of the data's validity is known as *integrity*.
- *Confidentiality*. Both authentication and integrity, however, leave the data itself alone, so if it's somehow intercepted, others can read it. The GSS-API therefore allows data to be encrypted, if underlying mechanisms support it. This encryption of data is known as *confidentiality*.

Mechanisms Available With GSS-API

The current implementation of the GSS-API works only with the Kerberos v5 security mechanism. (This includes its Sun variant, the Solaris Enterprise Authentication Mechanism, or SEAM. See "Introduction to SEAM" in *System Administration Guide: Security Services* for more information.) Kerberos v5 or SEAM must, therefore, be installed and running on any system on which GSS-API-aware programs are running.

RPCSEC_GSS Layer

Programmers who employ the RPC (Remote Procedure Call) protocol for their networking applications can use RPCSEC_GSS to provide security. RPCSEC_GSS is a separate layer that sits on top of GSS-API; it provides all the functionality of GSS-API in a way that is tailored to RPC. In fact, it serves to hide many aspects of GSS-API from the programmer, making RPC security especially accessible and portable. For more information on RPCSEC_GSS, see the *ONC+ Developer's Guide*.

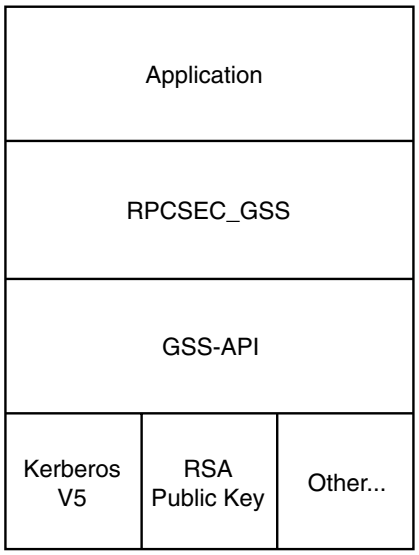

FIGURE 1–2 RPCSEC_GSS and GSS-API

What the GSS-API Does *Not* Do For You

Although the GSS-API makes protecting data simple, it does not do certain things, in order to maximize its generic nature. These include:

- Provide security credentials for a user or application. These must be provided by the underlying security mechanism(s). The GSS-API *does* allow applications to acquire credentials, either automatically or explicitly.
- Transfer data between applications. It is the application's responsibility to handle the transfer of *all* data between peers, whether it is security-related or "plain" data.
- Distinguish between different types of transmitted data (for example, to know or determine that a data packet is plain data and not GSS–API related).
- Indicate status due to remote (asynchronous) errors.
- Automatically protect information sent between processes of a multiprocess program.
- Allocate string buffers to be passed to GSS-API functions. See "Strings and Similar Data" on page 16.
- Deallocate GSS-API data spaces. These must be explicitly deallocated with functions such as `gss_release_buffer()` and `gss_delete_name()`.

Language Bindings

This document currently covers only the C language bindings (functions and data types) for the GSS-API. At some point a Java-bindings version of the GSS-API might become available.

Where to Get More Information

Two documents provide descriptions of the GSS-API (and are somewhat more oriented toward the GSS-API implementor than to the application developer). The Generic Security Service Application Program Interface document (`ftp://ftp.isi.edu/in-notes/rfc2743.txt`) provides a conceptual overview of the GSS-API, while the Generic Security Service API Version 2: C-Bindings document (`ftp://ftp.isi.edu/in-notes/rfc2744.txt`) discusses the specifics of the C-language-based GSS-API.

Some Introductory Concepts

Before looking at the actual process of using the GSS-API, let's examine four important concepts. They are: principals, GSS-API data types, status codes, and tokens.

Principals

In network-security terminology, a *principal* is a user, a program, or a machine. Principals can be either clients or servers. Examples of principals are: a user (*joe@machine*) logging into another machine; a network service (*nfs@machine*); a machine that runs an application (*swim2birds@eng.company.com*).

In the GSS-API, principals are referred to by a special data type— see "Names" on page 17.

GSS-API Data Types

The following sections explain the more important and visible GSS-API data types; see "GSS-API Data Types and Values" on page 122 for more information.

Caution – It is the responsibility of the calling application to free all data space that has been allocated.

Integers

Because the size of an `int` can vary from platform to platform, the GSS-API provides the following integer data type:

```
OM_uint32
```

which is a 32–bit unsigned integer.

Strings and Similar Data

Since the GSS-API handles all data in internal formats, strings must be converted to a GSS-API format before being passed to GSS-API functions. The GSS-API handles strings with the `gss_buffer_desc` structure; `gss_buffer_t` is a pointer to such a structure.

```
typedef struct gss_buffer_desc_struct {
     size_t     length;
     void       *value;
}  gss_buffer_desc *gss_buffer_t;
```

Therefore, strings must be put into a `gss_buffer_desc` structure before being passed to functions that use them. Consider a generic GSS-API function that takes a message and processes it in some way (for example, applies protection to it before it is transmitted), as follows:

EXAMPLE 1–1 Using Strings

```
char *message_string;
gss_buffer_desc input_msg_buffer;

input_msg_buffer.value = message_string;
input_msg_buffer.length = strlen(input_msg_buffer.value) + 1;

gss_generic_function(arg1, &input_msg_buffer, arg2...);

gss_release_buffer(input_msg_buffer);
```

Note that `input_msg_buffer` must be deallocated with `gss_release_buffer()` when you are finished with it.

The `gss_buffer_desc` object is not just for character strings; for example, tokens are manipulated as `gss_buffer_desc` objects. (See "GSS-API Tokens" on page 26.)

Names

A *name* refers to a principal — that is, a person, a machine, or an application, such as *joe@company* or *nfs@machinename*. In the GSS-API, names are stored as a `gss_name_t` object, which is opaque to the application. Names are converted from `gss_buffer_t` objects to the `gss_name_t` form by the `gss_import_name()` function. Every imported name has an associated *name type*, which indicates what kind of format the name is in. (See under "OIDs" on page 24 for more about name types, and see "Name Types" on page 123 for a list of valid name types).

`gss_import_name()` looks like this:

```
OM_uint32 gss_import_name (
       OM_uint32          *minor_status,
       const gss_buffer_t input_name_buffer,
       const gss_OID      input_name_type,
       gss_name_t         *output_name)
```

minor_status — Status code returned by the underlying mechanism. (See "Status Codes" on page 25.)

input_name_buffer — The `gss_buffer_desc` structure containing the name to be imported. The application must allocate this explicitly (see "Strings and Similar Data" on page 16 as well as Example 1–2.) This argument must be deallocated with `gss_release_buffer()` when the application is finished with it.

input_name_type — A `gss_OID` that specifies the format that the *input_name_buffer* is in. (See "Name Types" on page 25; also, "Name Types" on page 123 contains a table of valid name types.)

output_name — The `gss_name_t` structure to receive the name.

Slightly modifying the generic example shown in Example 1–1, here is how you can use `gss_import_name()`. First, the regular string is inserted into a `gss_buffer_desc` structure, and then `gss_import_name()` places it into a `gss_name_t` structure.

EXAMPLE 1–2 Using `gss_import_name()`

```
char *name_string;
gss_buffer_desc input_name_buffer;
gss_name_t      output_name_buffer;

input_name_buffer.value = name_string;
input_name_buffer.length = strlen(input_name_buffer.value) + 1;

gss_import_name(&minor_status, input_name_buffer,
                    GSS_C_NT_HOSTBASED_SERVICE, &output_name);

gss_release_buffer(input_name_buffer);
```

An imported name can be put back into a `gss_buffer_t` object for display in human-readable form with `gss_display_name()`; however, `gss_display_name()` does not guarantee that the resulting string will be the same as the original, because of the way the underlying mechanisms store names. The GSS-API includes several other functions for manipulating names; see "GSS-API Functions" on page 115.

A `gss_name_t` structure can contain several versions of a single name — one version produced for each mechanism supported by the GSS-API. That is, a `gss_name_t` structure for "joe@company" might contain one version of that name as rendered by Kerberos v5, another version as given by a different mechanism, and so on. The GSS-API provides a function, `gss_canonicalize_name()`, that takes as its input an internal name (that is, a `gss_name_t` structure) and a mechanism and yields a second internal name (also a `gss_name_t`) that contains only a single version of the name, specific to that mechanism.

Such a mechanism-specific name is called a Mechanism Name (MN). "Mechanism Name" is a slightly confusing label, since it refers not to the name of a mechanism, but to the name of a principal as produced by a given mechanism. This process is illustrated by Figure 1–3.

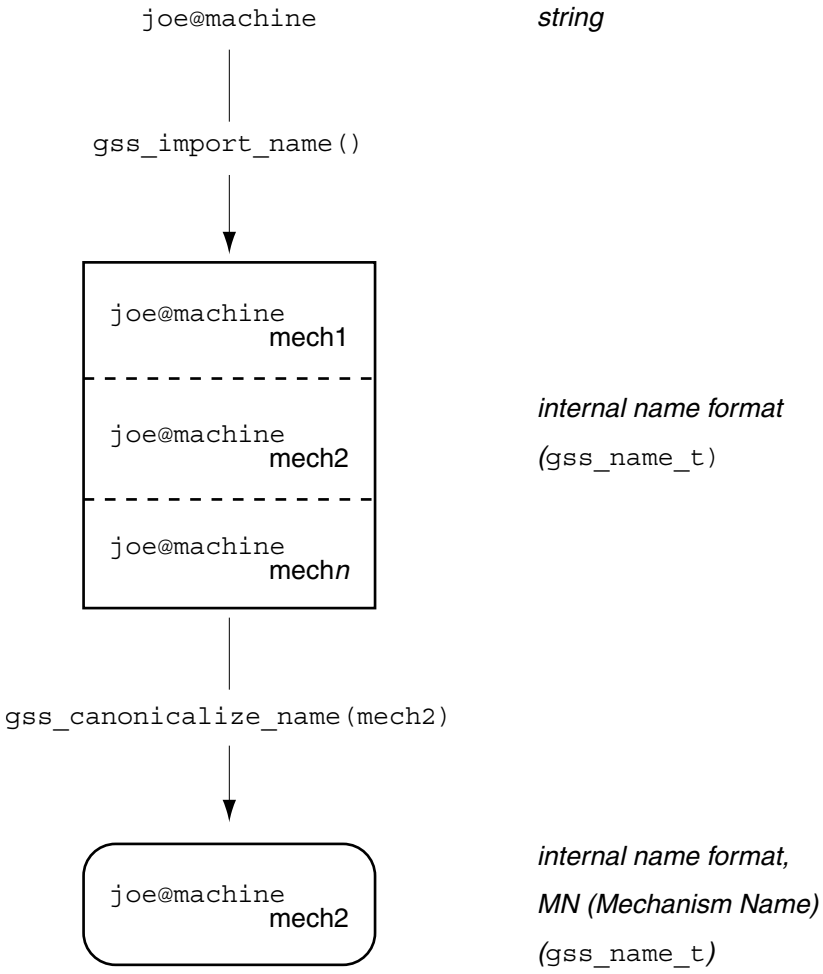

FIGURE 1–3 Internal Names and Mechanism Names

Comparing Names

Why is such a function useful? Consider the case where a server has received a name from a client and wants to look up that name in an Access Control List. (An Access Control List, or ACL, is a list of principals with particular access permissions.) One way to do this would be as follows:

1. Import the client name into GSS-API internal format with `gss_import_name()`, if it hasn't already been imported.

   In some cases, the server will receive a name in internal format, so this step will not be necessary — in particular, if the server is looking up the client's own name. (During context initiation, the client's own name is passed in internal format.)
2. Import each name in the ACL with `gss_import_name()`.
3. Compare each imported ACL name with the imported client's name, using `gss_compare_name()`.

This process is shown in Figure 1–4; in this case, we assume that Step 1 is needed.

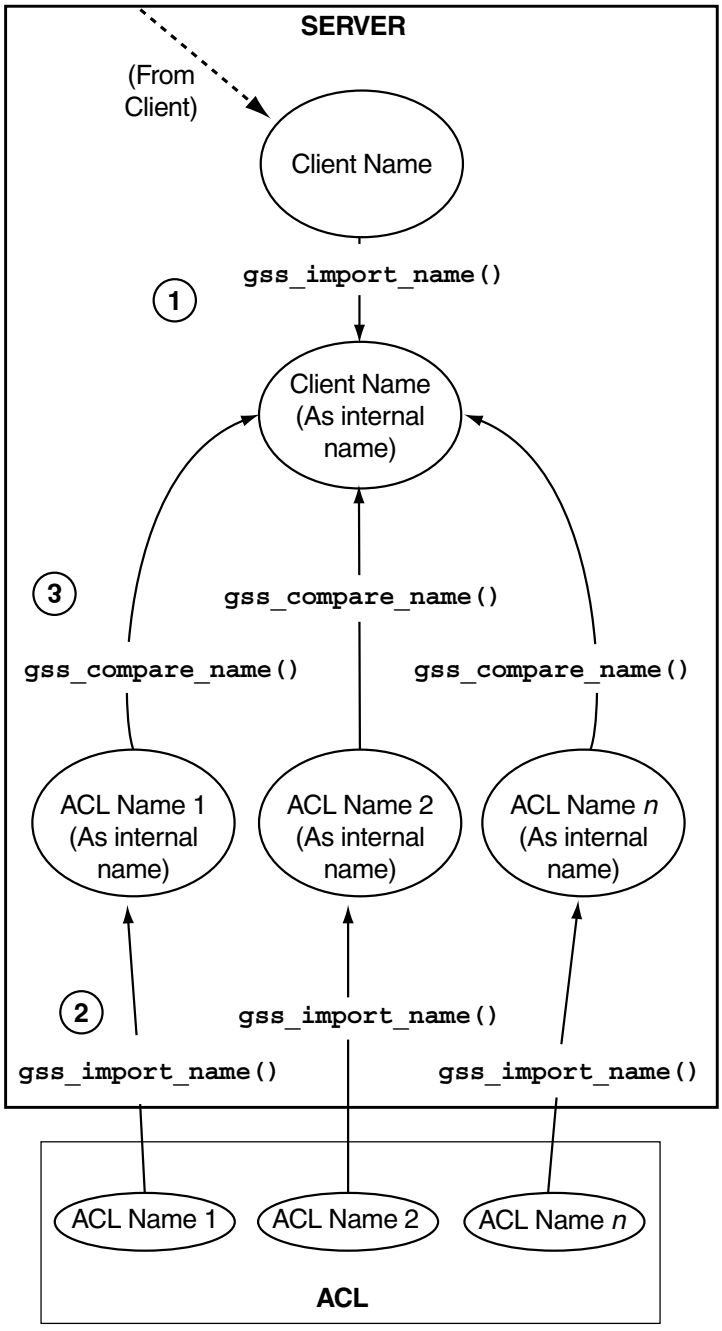

FIGURE 1–4 Comparing Names (Slow)

That procedure is fine if you only need to compare the client's name with a few names. However, it is a very slow way to check a large list! Running `gss_import_name()` and `gss_compare_name()` for every name in the ACL might require a lot of CPU cycles. This is a better way:

1. Import the client's name with `gss_import_name()` (if it hasn't already been imported).

   As with the previous method of comparing names, in some cases the server receives a name in internal format and so this step is not necessary.
2. Use `gss_canonicalize_name()` to produce an MN of the client's name.
3. Use `gss_export_name()` to produce an "exported name," a contiguous-string version of the client's name.
4. Compare the exported client's name with each name in the ACL by using `memcmp()`, which is a fast, low-overhead function.

This process is shown in Figure 1–5; again, assume the server needs to import the name received from the client.

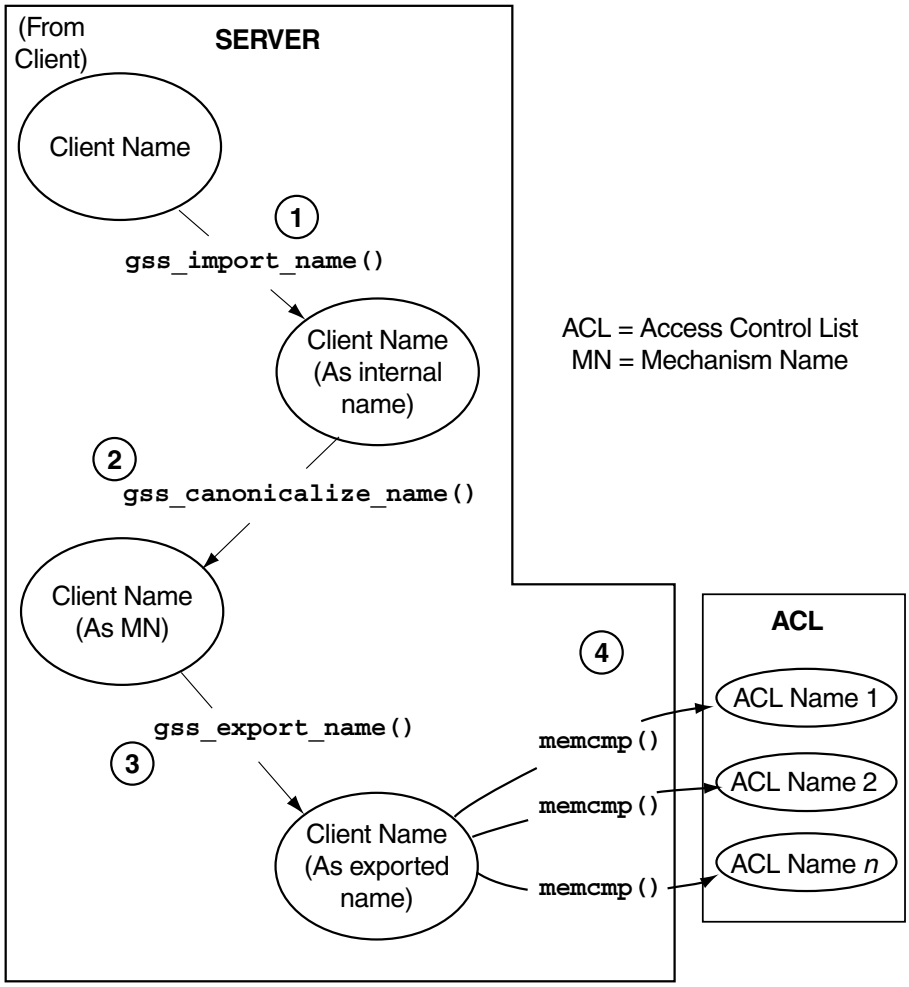

FIGURE 1–5 Comparing Names (Fast)

Because `gss_export_name()` expects a Mechanism Name (MN), you *must* run `gss_canonicalize_name()` on the client's name first.

See the `gss_canonicalize_name(3GSS)`, `gss_export_name(3GSS)`, and `gss_import_name(3GSS)` man pages for more information.

OIDs

Object Identifiers (OIDs) are used to store the following kinds of data: security mechanisms, QOPs (Quality of Protection values), and name types. OIDs are stored in the GSS-API `gss_OID_desc` structure; the GSS-API provides a pointer to the structure, `gss_OID`, as shown here.

EXAMPLE 1–3 OIDs

```
typedef struct gss_OID_desc_struct {
        OM_uint32   length;
        void        *elements;
     } gss_OID_desc, *gss_OID;
```

Further, one or more OIDs might be contained in a `gss_OID_set_desc` structure.

EXAMPLE 1–4 OID Sets

```
typedef struct gss_OID_set_desc_struct {
        size_t    count;
        gss_OID   elements;
     } gss_OID_set_desc, *gss_OID_set;
```

Caution – Applications should not attempt to deallocate OIDs with `free()`.

Mechanisms and QOPs

Although the GSS-API allows applications to choose which underlying security mechanism to use, applications should use the default mechanism selected by the GSS-API if possible. Likewise, the GSS-API allows an application to specify the QOP it wants for protecting data — a QOP (Quality of Protection) is the algorithm used for encrypting data or generating a cryptographic identification tag — the default QOP should be used if possible. The default mechanism is represented by passing the value GSS_C_NULL_OID to functions that expect a mechanism or QOP as an argument.

Caution – Specifying a security mechanism or QOP explicitly more or less defeats the purpose of using the GSS-API, because it limits the portability of an application. Other implementations of the GSS-API may not support that QOP or mechanism, or they may support it in limited or unexpected ways. Nonetheless, Appendix C briefly discusses how to find out which mechanisms and QOPs are available, and how to choose one.

Name Types

Besides QOPs and security mechanisms, OIDs are also used to indicate name types, which indicate the format for an associated name. For example, the function `gss_import_name()`, which converts the name of a principal from a string to a `gss_name_t` type, takes as one argument the format of the string to be converted. If the name type is (for example) GSS_C_NT_HOSTBASED_SERVICE, then the function knows that the name being input is of the form "service@host", as in "nfs@swim2birds"; if it's equal to, for instance, GSS_C_NT_EXPORT_NAME, then the function knows that it's a GSS-API exported name. Applications can find out which name types are available for a given mechanism with the `gss_inquire_names_for_mech()` function. A list of name types used by the GSS-API is given in "Name Types" on page 123.

Status Codes

All GSS-API functions return two types of codes that provide information on the function's success or failure. Both types of status codes are returned as `OM_uint32` values. The two types of return codes are as follows:

- *Major-status codes*. These are codes that indicate: a) generic GSS-API routine errors (such as giving a routine an invalid mechanism); b) calling errors specific to a particular GSS-API language binding (namely, a function argument that cannot be read, cannot be written, or is malformed); or c) both. Additionally, major-status codes can provide supplementary information about a routine's status — that an operation is not finished, for example, or that a token has been sent out of order. If no errors occur, the routine returns a major status value of `GSS_S_COMPLETE`.

  Major-status codes are returned as shown here:

  ```
  OM_uint32 major_status ;    /* status returned by GSS-API */

  major_status = gss_generic_function(arg1, arg2 ...);
  ```

  Major status return codes can be processed like any other `OM_uint32`. For example:

  ```
  OM_uint32 maj_stat;

  maj_sta = gss_generic_function(arg1, arg2 ...);

  if (maj_stat == GSS_CREDENTIALS_EXPIRED)
          <do something...>
  ```

  They can be processed with the macros `GSS_ROUTINE_ERROR()`, `GSS_CALLING_ERROR()`, and `GSS_SUPPLEMENTARY_INFO()`. "GSS-API Status Codes" on page 118 explains how to read major-status codes and contains a list of GSS-API status codes.

- *Minor status codes*. These are returned by the underlying mechanism, and so are not specifically documented in this manual.

  Every GSS-API function has as its first argument an `OM_uint32` for the minor code status. The minor status code is stored here when the function returns to the function that called it:

```
OM_uint32 *minor_status ;    /* status returned by mech */

major_status = gss_generic_function(&minor_status, arg1, arg2 ...);
```

  The *minor_status* parameter is always set by a GSS-API routine, even if it returns a fatal major-code error, although most other output parameters can remain unset. However, output parameters that are expected to return pointers to storage allocated by the routine are set to NULL to indicate that no storage was actually allocated. Any length field associated with such pointers (as in a `gss_buffer_desc` structure) are set to zero. In these cases applications don't need to release these buffers.

GSS-API Tokens

The basic unit of currency, so to speak, in the GSS-API is the *token*. Applications using the GSS-API communicate with each other by using tokens, both for exchanging data and for making security arrangements. Tokens are declared as `gss_buffer_t` data types and are opaque to applications.

The two types of tokens are: *context-level tokens* and *per-message tokens*. Context-level tokens are used primarily when a context is established (initiated and accepted), although they can also be passed afterward to manage a context.

Per-message tokens are used after a context has been established, and are used to provide protection services on data. For example, if an application wants to send a message to another application, it might use the GSS-API to generate a cryptographic identifier to go along that message; that identifier would be stored in a token.

Per-message tokens can be considered with regard to "messages" as follows. A *message* is a piece of data that an application sends to a peer; for example, the `ls` command sent to an `ftp` server. A *per-message token* is an object generated by the GSS-API for that message, such as a cryptographic tag, or the encrypted form of the message. (Semantically speaking, this last example is mildly inaccurate: an encrypted message is still a message, not a token, since a token is *only* the GSS-API-generated information. However, informally, *message* and *per-message* token are often used interchangeably.)

It is the responsibility of the application (not the GSS-API) to:

1. Send and receive tokens. The developer usually needs to write generalized read and write functions for performing these actions. "`send_token()`" on page 112 and "`recv_token()`" on page 113 are examples of such functions.
2. Distinguish between types of tokens and manipulate them accordingly.

   Because tokens are opaque to applications, there is no difference (to the application) between one token and another. Therefore, an application must be able to distinguish one token from another without explicitly knowing their contents, before passing them on to the appropriate GSS-API functions. The ways an application can distinguish tokens include:

   - By state — that is, through the control-flow of a program. For example, if an application is waiting to accept a context, it can assume that any token it receives is a context-level token related to context-establishment, because it expects peers to wait until the context is fully established before sending message (data) tokens. After the context is established, the application can assume that any tokens it receives are message tokens. This is a fairly common way to handle tokens; the sample programs in this book use this method.
   - An application might distinguish types of tokens when sending and receiving them. For example, if the application has its own function for sending tokens to peers, it can include a flag indicating what kind of token is being sent:

     ```
     gss_buffer_t token;     /* declare the token */
     OM_uint32 token_flag       /* flag for describing the type of token */

         <get token from a GSS-API function>

     token_flag = MIC_TOKEN;     /* specify what kind of token it is */
     send_a_token(&token, token_flag);
     ```

     Then the receiving application would have a receiving function (say, "`get_a_token()`") that would check the *token_flag* argument.

   - A third way might be through explicit tagging; for example, applications might use their own "meta-tokens": user-defined structures that contain tokens received from GSS-API functions, along with user-defined fields that signal how the GSS-API-provided tokens are to be used.

Interprocess Tokens

The GSS-API permits a security context to be passed from one process to another in a multiprocess application. Typically, this application has accepted a client's context and wants to share it among its processes. See "Context Export and Import" on page 51 for information on multiprocess applications.

The `gss_export_context()` function creates an interprocess token that contains information allowing the context to be reconstituted by a second process. It is the responsibility of the application to pass this interprocess token from one process to the other, just as it is the application's responsibility to pass tokens to other applications.

Since this interprocess token might contain keys or other sensitive information, and since it cannot be guaranteed that all GSS-API implementations will cryptographically protect interprocess tokens, it is up to the application to protect them before exchange. This may involve encrypting them with `gss_wrap()`, if encryption is available.

Note – Interprocess tokens cannot be assumed to be transferable across different GSS-API implementations.

Programming Using the GSS-API

This section is designed to show, in general steps, how to implement secure data exchange using the GSS-API. It does not explain every GSS-API function. Instead, it concentrates on the half-dozen or so functions that are most central to using the GSS-API. For more information, see Appendix B, which contains a list of all GSS-API functions (as well as GSS-API status codes and data types). Additionally, you can find out more about any GSS-API function by checking its man page.

To make things easier, this manual follows a simple model: A client application sends data to a remote server. The client does so directly — that is, without mediation by transport protocol layers such as RPC. A set of sample programs (client and server) are shown in Appendix A. Chapter 2 takes you step-by-step through these programs.

Overview

These are the basic steps in using the GSS-API:

1. Each application, sender and recipient, acquires credentials explicitly, if credentials have not been acquired automatically.
2. The sender initiates a security context and the recipient accepts it.
3. The sender applies security protection to the message (data) it wants to transmit. This means that it either encrypts the message or stamps it with an identification tag. The sender transmits the protected message.

(The sender can choose not to apply either security protection, in which case the message has only the default GSS-API security service associated with it. That is authentication, in which the recipient knows that the sender is who it claims to be.)

4. The recipient decrypts the message (if needed) and verifies it (if appropriate).
5. (Optional) The recipient returns an identification tag to the sender for confirmation.
6. Both applications destroy the shared security context. If necessary, they can also deallocate any "leftover" GSS-API data.

Applications that use the GSS-API should `include` the file `gssapi.h`.

A general schema of this process is presented in Figure 1–6, which shows one way that the GSS-API can be used; other scenarios are possible.

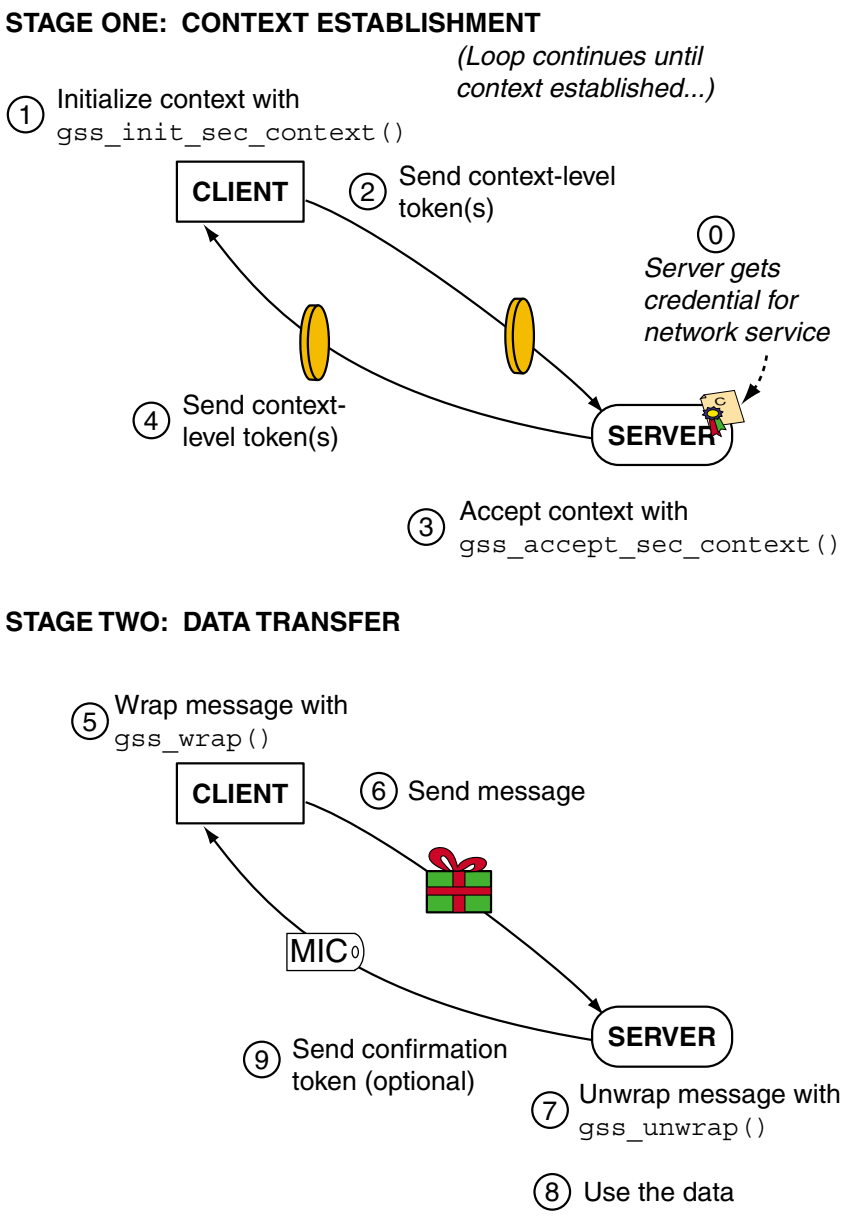

FIGURE 1–6 Using the GSS-API: An Overview

Credentials

A *credential* is a data structure that provides proof of an application's claim to a principal name. An application uses a credential to establish its global identity. You can think of a credential as a principal's "identity badge," a set of information that proves that a person, machine, or program is who it claims to be and, often, what privileges it has.

The GSS-API does not provide credentials. Credentials are created by the security mechanisms that underly the GSS-API, before GSS-API functions are called. In many cases, for example, users receive credentials when they log in to a system.

A given GSS-API credential is valid for a single principal. A single credential can contain several elements for that principal, each created by a different mechanism, as shown in Figure 1–7. This means that a credential acquired on a machine with several security mechanisms will be valid if transferred to a machine that has only a subset of those mechanisms. The GSS-API accesses credentials through the `gss_cred_id_t` structure; this structure is called a *credential handle*. Credentials are opaque to applications; you don't need to know the specifics of a given credential.

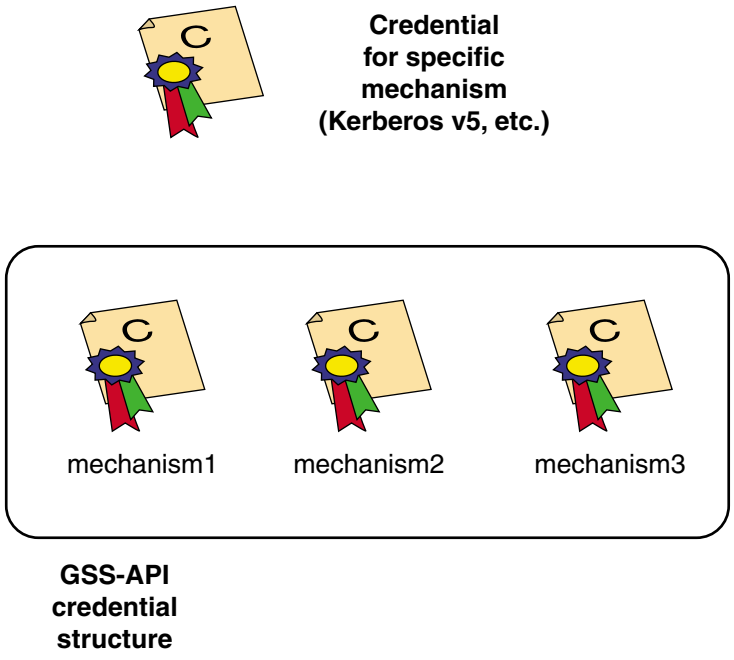

FIGURE 1–7 Generalized GSS-API Credential

Credentials come in three forms:

- GSS_C_INITIATE: A credential of this type identifies applications that only initiate security contexts.
- GSS_C_ACCEPT: A credential of this type identifies applications that only accept security contexts.
- GSS_C_BOTH: A credential of this type identifies applications that can initiate or accept security contexts.

Acquiring Credentials

Before a security context can be established, both the server and the client must acquire their respective credentials. Once acquired, credentials can be re-used until they expire, at which time they must be re-acquired. Credentials used by the client and those used by the server might have different lifetimes.

GSS-API-based applications acquire credentials in one of two ways:

- By using the `gss_acquire_cred()` function (in some cases this will be the `gss_add_cred()` function), or
- By specifying a default credential, represented by the value GSS_C_NO_CREDENTIAL, when establishing a context.

In most cases, `gss_acquire_cred()` is called only by a context acceptor (server), because the context initiator (client) is likely to have received its credentials at login. The initiator, therefore, can usually specify only the default credential. The context acceptor can also bypass using `gss_acquire_cred()` and use its default credential instead.

The initiator's credential proves its identity to other processes, while the acceptor acquires a credential to enable it to accept a security context. Consider the case of a client making an ftp request to a server. The client already has a credential, from login, and the GSS-API is automatically retrieves that credential when the client attempts to initiate a context. The server program, however, explicitly acquires credentials for the requested service (ftp).

`gss_acquire_cred()` takes the syntax shown below:

```
OM_uint32 gss_acquire_cred (
OM_uint32          *minor_status,
const gss_name_t   desired_name,
OM_uint32          time_req,
const gss_OID_set  desired_mechs,
gss_cred_usage_t   cred_usage,
gss_cred_id_t      *output_cred_handle,
gss_OID_set        *actual_mechs,
OM_uint32          *time_rec)
```

minor_status	The status code given by the underlying mechanism upon return.
desired_name	The name of the principal whose credential should be acquired. In our example above, it would be "ftp." This argument would be created with `gss_import_name()` (see "Names" on page 17). Note that if *desired_name* is set to GSS_C_NO_NAME, then the credential returned is a generic one that causes the GSS-API context-initiation and context-acceptance routines to use their default behavior with regard to credentials. In other words, passing GSS_C_NO_NAME to `gss_acquire_cred()` returns a credential that, when passed to `gss_init_sec_context()` or `gss_accept_sec_context()`, is equivalent to passing them the default credential request (GSS_C_NO_CREDENTIAL). See "Context Initiation (Client)" on page 34 and "Context Acceptance (Server)" on page 40 for more information.
time_req	The length of time (in seconds) for which the credential should remain valid. Specify GSS_C_INDEFINITE to request the maximum permitted lifetime.
desired_mechs	The set of underlying mechanisms that the application wants to use with this credential. This is a `gss_OID_set` data structure containing one or more `gss_OID` structures, each representing an appropriate mechanism. If possible, specify GSS_C_NO_OID_SET to get a default set from the GSS-API.
cred_usage	A flag that indicates how this credential should be used: for context initiation (GSS_C_INITIATE), acceptance (GSS_C_ACCEPT), or both (GSS_C_BOTH).
output_cred_handle	The credential handle returned by this function.
actual_mechs	A set of mechanisms that can be used with this credential. If you don't need to know what the mechanisms are, set this to NULL.
time_rec	The actual number of seconds for which the credential is valid. Set to NULL if this value doesn't interest you.

`gss_acquire_cred()` returns `GSS_S_COMPLETE` if it completes successfully. If it cannot return a valid credential, it returns `GSS_S_NO_CRED`; see the `gss_acquire_cred(3GSS)` man page for other error codes. An example of acquiring a credential can be found in "Acquiring Credentials" on page 75 (program listing in "`server_acquire_creds()`" on page 99).

`gss_add_cred()` is similar to `gss_acquire_cred()`, but allows an application to either create a new credential handle based on an existing credential or to add a new credential element to the existing one. If GSS_C_NO_CREDENTIAL is specified as the existing credential, then `gss_add_cred()` will create a new credential based on default behavior. See the `gss_add_cred(3GSS)` man page for more information.

Context Establishment

As stated earlier, the two most significant things that the GSS-API does in providing security are to create security contexts and to protect data. After an application has the credential(s) it needs, it's time to establish a security context. To do this, one application (typically a client) initiates the context, and another (usually a server) accepts it. Multiple contexts between peers are allowed.

The communicating applications establish a joint security context by exchanging authentication tokens. The security context is a pair of GSS-API data structures that contain information shared between the two applications. This information describes the state of each application (in terms of security). A security context is required for protection of data.

Context Initiation (Client)

Security context initiation between an application and a remote peer is done using the `gss_init_sec_context()` function. If successful, this function returns a *context handle* for the context being established and a context-level token to send to the acceptor. Before calling `gss_init_sec_context()`, however, the client should:

1. Acquire credentials, if necessary, with `gss_acquire_cred()`. Commonly, however, the client has received credentials at login, and can skip this step.
2. Import the name of the server into GSS-API internal format with `gss_import_name()`. See "Names" on page 17 for more about names and `gss_import_name()`.

When calling `gss_init_sec_context()`, the client typically passes the following argument values:

- GSS_C_NO_CREDENTIAL for the *cred_handle* argument, to indicate the default credential.
- GSS_C_NULL_OID for the *mech_type* argument, to indicate the default mechanism.
- GSS_C_NO_CONTEXT for the *context_handle* argument, to indicate an initial null context. Because `gss_init_sec_context()` is usually called in a loop, subsequent calls should pass the context handle returned by previous calls.

- GSS_C_NO_BUFFER for the *input_token* argument, to indicate an initially empty token. Alternatively, the application can pass a pointer to a `gss_buffer_desc` object whose length field has been set to zero.
- The name of the server, imported into internal GSS-API format with `gss_import_name()`.

Applications are not bound to use these default values. Additionally, the client will specify its requirements for other security parameters with the *req_flags* argument. The full set of `gss_init_sec_context()` arguments is described below.

The context acceptor can require several "handshakes" in order to establish a context; that is, it can require the initiator to send more than one piece of context information before it considers the context fully established. Therefore, for portability, context initiation should always be done as part of a loop that checks whether the context has been fully established.

If the context is not complete, `gss_init_sec_context()` returns a major-status code of `GSS_C_CONTINUE_NEEDED`. Thus a loop should use `gss_init_sec_context()`'s return value to test whether to continue the initiation loop.

The client passes context information to the server in the form of the *output token* returned by `gss_init_sec_context()`. The client receives information back from the server as an *input token*, which can then be passed as an argument to subsequent calls of `gss_init_sec_context()`. If the received input token has a length of zero, however, then no more output tokens are required by the server.

Therefore, in addition to checking for the return status of `gss_init_sec_context()`, the loop should check the input token's length to see if a further token needs to be sent to the server. (Before the loop begins, the input token's length should be initialized to zero, either by setting the input token to GSS_C_NO_BUFFER or by setting the structure's length field to a value of zero.)

This is what such a loop can look like, highly generalized:

```
context = GSS_C_NO_CONTEXT
input token = GSS_C_NO_BUFFER

do

    call gss_init_sec_context(credential, context, name, input token,
                              output token, other args...)

    if (there's an output token to send to the acceptor)
        send the output token to the acceptor
        release the output token

    if (the context is not complete)
```

receive an input token from the acceptor

if (*there's a GSS-API error*)
delete the context

until the context is complete

Naturally, a real loop will be more complete, with, for example, much more extensive error-checking. See "Establishing a Context" on page 68 (program listing in "`client_establish_context()`" on page 91) for a real example of such a context-initiation loop. Additionally, the `gss_init_sec_context(3GSS)` man page gives a less generic example than this.

Again, the GSS-API does not send or receive tokens; these must be handled by the application. Examples of token-transferring functions are found in "`send_token()`" on page 112 and "`recv_token()`" on page 113.

Here is a synopsis of `gss_init_sec_context()`. For more information, see the `gss_init_sec_context(3GSS)` man page.

EXAMPLE 1–5 `gss_init_sec_context()`

```
OM_uint32 gss_init_sec_context (
  OM_uint32                   *minor_status,
  const gss_cred_id_t         initiator_cred_handle,
  gss_ctx_id_t                *context_handle,
  const gss_name_t            target_name,
  const gss_OID               mech_type,
  OM_uint32                   req_flags,
  OM_uint32                   time_req,
  const gss_channel_bindings_t input_chan_bindings,
  const gss_buffer_t          input_token
  gss_OID                     *actual_mech_type,
  gss_buffer_t                output_token,
  OM_uint32                   *ret_flags,
  OM_uint32                   *time_rec )
```

minor_status	The status code returned by the underlying mechanism.
initiator_cred_handle	The credential handle for the application. This should be initialized to GSS_C_NO_CREDENTIAL to indicate the default credential to use.
context_handle	The context handle to be returned. This should be set to GSS_C_NO_CONTEXT before the loop begins.
target_name	The name of the principal to connect to; for example, "nfs@machinename."
mech_type	The security mechanism to use. Set this to GSS_C_NO_OID to get the default provided by the GSS-API.

req_flags

Flags indicating additional services or parameters requested for this context. *req_flags* flags should be logically `OR`'d to make the desired bit-mask value, as in:

```
GSS_C_MUTUAL_FLAG | GSS_C_DELEG_FLAG
```

GSS_C_DELEG_FLAG

Requests that delegation of the initiator's credentials be permitted. See "Delegation" on page 45.

GSS_C_MUTUAL_FLAG

Requests mutual authentication. See "Mutual Authentication" on page 46.

GSS_C_REPLAY_FLAG

Requests detection of repeated messages. See "Out-of-Sequence Detection and Replay Detection" on page 47.

GSS_C_SEQUENCE_FLAG

Requests detection of out-of-sequence messages. See "Out-of-Sequence Detection and Replay Detection" on page 47.

GSS_C_CONF_FLAG

Requests that the confidentiality service be allowed for transferred messages; that is, that messages be encrypted. If confidentiality is not allowed, then only data-origin authentication and integrity services can be applied (this last only if GSS_C_INTEG_FLAG is not false).

GSS_C_INTEG_FLAG

Requests that the integrity service be applicable to messages; that is, that messages may be stamped with a MIC to ensure their validity.

GSS_C_ANON_FLAG

Requests that the initiator remain anonymous. See "Anonymous Authentication" on page 49.

time_req

The number of seconds for which the context should remain valid. Set this to zero (0) to request the default.

input_chan_bindings Specific peer-to-peer channel identification information connected with the security context. See "Channel Bindings" on page 49 for more information about channel bindings. Set to GSS_C_NO_CHANNEL_BINDINGS if you don't want to use channel bindings.

input_token Token received from the context acceptor, if any. Should be initialized to GSS_C_NO_BUFFER before the function is called (or its length field set to zero).

actual_mech_type The mechanism actually used in the context. Specify NULL if you don't need to know.

output_token The token to send to the acceptor.

ret_flags Flags indicating additional services or parameters requested for this context. *ret_flags* flags should be logically `AND`'d to test the returned bit-mask value, as in:

```
if (ret_flags & GSS_C_CONF_FLAG)
     confidentiality = TRUE;
```

GSS_C_DELEG_FLAG

If true, indicates that the initiator's credentials can be delegated. See "Delegation" on page 45.

GSS_C_MUTUAL_FLAG

If true, indicates that mutual authentication is allowed. See "Mutual Authentication" on page 46.

GSS_C_REPLAY_FLAG

If true, indicates that detection of repeated messages is in effect. See "Out-of-Sequence Detection and Replay Detection" on page 47.

GSS_C_SEQUENCE_FLAG

If true, indicates that detection of out-of-sequence messages is in effect. See "Out-of-Sequence Detection and Replay Detection" on page 47.

GSS_C_CONF_FLAG

If true, confidentiality service is allowed for transferred messages; that is, that messages can be encrypted. If confidentiality is not allowed, then only data-origin authentication, and integrity services can be applied (this last only if `GSS_C_INTEG_FLAG` is not returned as false).

GSS_C_INTEG_FLAG

If true, the integrity service can be applied to messages; that is, that messages can be stamped with a MIC to ensure their validity.

GSS_C_ANON_FLAG

If true, indicates that the context initiator will remain anonymous. See "Anonymous Authentication" on page 49.

GSS_C_PROT_READY_FLAG

Sometimes context establishment can take several passes, and sometimes the client might have to wait before it's complete. Even though a context is not fully established, `gss_init_sec_context()` can indicate what protection services, if any, will be available after the context is complete. An application can therefore buffer its data, sending it when the context is eventually fully established.

If *ret_flags* indicates `GSS_C_PROT_READY_FLAG`, the protection services indicated by the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG` flags are available even if the context has not been fully established (that is, if `gss_init_sec_context()` returns `GSS_S_CONTINUE_NEEDED`). An application can then call the appropriate wrapping functions, `gss_wrap()` or `gss_get_mic()`, with the preferred protection services, and buffer the output for transfer when the context is complete.

If `GSS_C_PROT_READY_FLAG` is false, then the application cannot make any assumptions about data protection, and must wait until the context is complete (that is, when `gss_init_sec_context()` returns `GSS_S_COMPLETE`).

Note – Earlier versions of the GSS-API did not support the `GSS_C_PROT_READY_FLAG` argument, so developers wanting to maximize portability should determine which per-message services are available by looking at the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG` flags after a context has been successfully established.

GSS_C_TRANS_FLAG

	This flag indicates whether this context can be exported. For more information on importing and exporting contexts, see "Context Export and Import" on page 51.
time_rec	Number of seconds for which the context will remain valid. Specify NULL if you're not interested in this value.

In general, the parameter values returned when a context is not fully established are those that would be returned when the context is complete. See the `gss_init_sec_context()` man page for more information.

`gss_init_sec_context()` returns `GSS_S_COMPLETE` if it completes successfully. If a context-establishment token is required from the peer application, it returns `GSS_S_CONTINUE_NEEDED`. If there are errors, it returns error codes, which can be found on the `gss_init_sec_context(3GSS)` man page.

If context initiation fails, the client should disconnect from the server.

Context Acceptance (Server)

The other half of context establishment is context acceptance, which is done through the `gss_accept_sec_context()` function. In a typical scenario, a server accepts a context initiated (with `gss_init_sec_context()`) by a client.

`gss_accept_sec_context()` takes as its main input an input token sent by the initiator. It returns a context handle as well as an output token to be returned to the initiator. Before `gss_accept_sec_context()` can be called, however, the server should acquire credentials for the service requested by the client. The server acquires these credentials with the `gss_acquire_cred()` function. Alternatively, the server can bypass acquiring credentials explicitly and instead specify the default credential (indicated by `GSS_C_NO_CREDENTIAL`) when calling `gss_accept_sec_context()`.

When calling `gss_accept_sec_context()`, the server passes the following argument values:

- The credential handle returned by `gss_acquire_cred()`, or GSS_C_NO_CREDENTIAL to indicate the default credential, for the *cred_handle* argument.
- GSS_C_NO_CONTEXT for the *context_handle* argument, to indicate an initial null context. Note that since `gss_init_sec_context()` is usually called in a loop, subsequent calls should pass the context handle returned by previous calls.
- The context token received from the client for the *input_token* argument.

The full set of `gss_accept_sec_context()` arguments is described in the following paragraphs.

Security context establishment may require several "handshakes"; that is, the initiator and acceptor may have to send more than one piece of context information before the context is fully established. Therefore, for portability, context acceptance should always be done as part of a loop that checks whether the context has been fully established. If it hasn't, `gss_accept_sec_context()` returns a major-status code of `GSS_C_CONTINUE_NEEDED`. Thus a loop should use the value returned by `gss_accept_sec_context()` to test whether to continue the acceptance loop.

The context acceptor returns context information to the context initiator in the form of the output token returned by `gss_accept_sec_context()`. Subsequently, the acceptor can receive further information from the initiator as an input token, which is then passed as an argument to subsequent calls of `gss_accept_sec_context()`. When `gss_accept_sec_context()` has no more tokens to send to the initiator, it returns an output token with a length of zero. Therefore, in addition to checking for the return status of `gss_accept_sec_context()`, the loop should check the output token's length to see if a further token must be sent. Before the loop begins, the output token's length should be initialized to zero, either by setting the output token to GSS_C_NO_BUFFER or by setting the structure's length field to a value of zero.

This is what such a loop might look like, highly generalized:

```
context = GSS_C_NO_CONTEXT
output token = GSS_C_NO_BUFFER

do

    receive an input token from the initiator

    call gss_accept_sec_context(context, cred handle, input token,
                                output token, other args...)

    if (there's an output token to send to the initiator)
        send the output token to the initiator
        release the output token

    if (there's a GSS-API error)
        delete the context

until the context is complete
```

Naturally, a real loop will be more complete, doing much more extensive error-checking. See "Accepting a Context" on page 77 (listing in "`server_establish_context()`" on page 103) for a real example of such a context-acceptance loop. Additionally, the `gss_accept_sec_context()` man page gives a somewhat less generic example than this.

Again, the GSS-API does not send or receive tokens; these must be handled by the application. Examples of token-transferring functions are found in "`send_token()` and `recv_token()`" on page 112.

Here is a synopsis of `gss_accept_sec_context()`. For more information, see the `gss_accept_sec_context(3GSS)` man page.

EXAMPLE 1–6 `gss_accept_sec_context()`

```
OM_uint32 gss_accept_sec_context (
  OM_uint32                   *minor_status,
  gss_ctx_id_t                *context_handle,
  const gss_cred_id_t         acceptor_cred_handle,
  const gss_buffer_t          input_token_buffer,
  const gss_channel_bindings_t input_chan_bindings,
  const gss_name_t            *src_name,
  gss_OID                     *mech_type,
  gss_buffer_t                output_token,
  OM_uint32                   *ret_flags,
  OM_uint32                   *time_req,
  gss_cred_id_t               *delegated_cred_handle)
```

minor_status — The status code returned by the underlying mechanism.

context_handle — The context handle to return to the initiator. This argument should be set to GSS_C_NO_CONTEXT before the loop begins.

acceptor_cred_handle — The handle for the credentials acquired by the acceptor, typically through `gss_acquire_cred()`). Can be initialized to GSS_C_NO_CREDENTIAL to indicate a default credential to use. If no default credential is defined, the function returns `GSS_C_NO_CRED`.

(Note: if `gss_acquire_cred()` was passed GSS_C_NO_NAME as a principal name, it produces a credential that will cause `gss_accept_sec_context()` to treat it as a default credential.)

input_token_buffer — Token received from the context initiator.

input_chan_bindings — Specific peer-to-peer channel identification information connected with the security context. See "Channel Bindings" on page 49 for more information about channel bindings. Set to GSS_C_NO_CHANNEL_BINDINGS if you don't want to use channel bindings.

src_name — The name of the initiating principal; for example, `nfs@machinename`. If you don't care, set to NULL.

mech_type — The security mechanism used. Set to NULL if you don't care which mechanism is used.

output_token

The token to send to the initiator. Should be initialized to GSS_C_NO_BUFFER before the function is called (or its length field set to zero). If the length is zero, no token needs to be sent.

ret_flags

Flags indicating additional services or parameters requested for this context. *ret_flags* flags should be logically `AND`'d to test the returned bit-mask value, as in:

```
if (ret_flags & GSS_C_CONF_FLAG)
     confidentiality = TRUE;
```

GSS_C_DELEG_FLAG

Indicates the initiator's credentials may be delegated via the *delegated_cred_handle* argument. See "Delegation" on page 45.

GSS_C_MUTUAL_FLAG

Indicates that mutual authentication is available. See "Mutual Authentication" on page 46.

GSS_C_REPLAY_FLAG

Indicates that detection of repeated messages is available. See "Out-of-Sequence Detection and Replay Detection" on page 47.

GSS_C_SEQUENCE_FLAG

Indicates that detection of out-of-sequence messages is available. See "Out-of-Sequence Detection and Replay Detection" on page 47.

GSS_C_CONF_FLAG

If true, confidentiality service is allowed for transferred messages; that is, that messages can be encrypted. If confidentiality is not allowed, then only data-origin authentication and integrity services can be applied (this last only if `GSS_C_INTEG_FLAG` is not returned as false).

GSS_C_INTEG_FLAG

If true, the integrity service can be applied to messages; that is, that messages can be stamped with a MIC to ensure their validity.

GSS_C_ANON_FLAG

Indicates that the context initiator will be anonymous. See "Anonymous Authentication" on page 49.

GSS_C_PROT_READY_FLAG

Sometimes context establishment can take several passes, and sometimes the client can send enough information on the initial passes to allow the acceptor to process context-related data, even though the context is incomplete. In those circumstances the acceptor needs to know in which way, if any, the information has been protected.

If true, `GSS_C_PROT_READY_FLAG` indicates that the protection services indicated by the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG` flags are available. The acceptor can therefore call the appropriate data–reception functions, `gss_unwrap()` or `gss_verify_mic()`, with these services in mind.

(Additionally, as with the context initiator, the acceptor can use these flags in buffering any data it might want to send to the initiator, transmitting it when the context is fully established.)

If `GSS_C_PROT_READY_FLAG` is false, then the acceptor cannot make any assumptions about data protection, and must wait until the context is complete (that is, when `gss_accept_sec_context()` returns `GSS_S_COMPLETE`).

Note – Earlier versions of the GSS-API did not support the `GSS_C_PROT_READY_FLAG` argument, so developers wanting to maximize portability should determine which per-message services are available by looking at the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG` flags after a context has been successfully established.

GSS_C_TRANS_FLAG

If true, this context can be exported. For more information on importing and exporting contexts, see "Context Export and Import" on page 51.

time_rec — Number of seconds for which the context will remain valid. Specify NULL if you're not interested in this value.

delegated_cred_handle — The credential handle for credentials received from the context initiator, that is, the client's credentials. Valid only if

the initiator has requested that the acceptor act as a proxy: that is, if the *ret_flags* argument resolves to `GSS_C_DELEG_FLAG`. See "Delegation" on page 45 for more about delegation.

`gss_accept_sec_context()` returns `GSS_S_COMPLETE` if it completes successfully. If the context is not complete, it returns `GSS_S_CONTINUE_NEEDED`. If there are errors, it returns error codes; for more information, see the `gss_accept_sec_context(3GSS)` man page.

Additional Context Services

The `gss_init_sec_context()` function (see "Context Initiation (Client)" on page 34) allows an application to request certain additional data protection services beyond basic context establishment. These services, discussed below, are requested through the *req_flags* argument to `gss_init_sec_context()`.

Because not all mechanisms offer all these services, `gss_init_sec_context()`'s *ret_flags* argument indicates which of them are available in a given context. Similarly, the context acceptor can determine which services are available by looking at the *ret_flags* value returned by the `gss_accept_sec_context()` function. The additional services are explained in the following sections.

Delegation

If permitted, a context initiator can request that the context acceptor act as a proxy, in which case the acceptor can initiate further contexts on behalf of the initiator. An example of such delegation would be where someone on Machine A wanted to `rlogin` to Machine B, and then `rlogin` from Machine B to Machine C, as shown in Figure 1–8. (Depending on the mechanism, the delegated credential identifies B either as A or "B acting for A.")

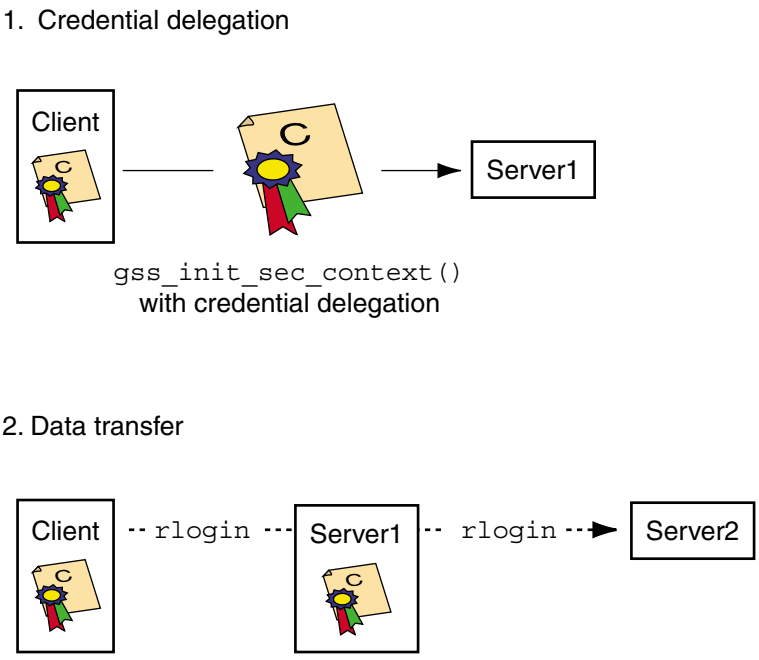

FIGURE 1–8 Credential Delegation

If delegation is permitted, *ret_flags* will be set to `GSS_C_DELEG_FLAG`; the acceptor receives a delegated credential as the *delegated_cred_handle* argument of `gss_accept_sec_context()`. Delegating a credential is not the same as exporting a context (see "Context Export and Import" on page 51). One difference is that an application can delegate its credentials multiple times simultaneously, while a context can only be held by one process at a time.

Mutual Authentication

If you are using `ftp`to download files into a public `ftp` site, you probably don't require that the site prove its identity, even if it requires proof of your own. On the other hand, if you are providing a password or credit card number to an application, you probably want to be sure of the receiver's *bona fides*. In these cases, *mutual authentication* is required — that is, both the context initiator and the acceptor must prove their identities.

A context initiator can request mutual authentication by setting `gss_init_sec_context()`'s *req_flags* argument to the value GSS_C_MUTUAL_FLAG. If mutual authentication has been authorized, the function indicates authorization by setting the *ret_flags* argument to this value. *If mutual authentication is requested but not available, it is the initiating application's responsibility to respond accordingly — the GSS-API will not terminate a context for this reason*. Some mechanisms will perform mutual authentication regardless of whether it has been requested.

Out-of-Sequence Detection and Replay Detection

In the common case where a context initiator is transmitting several sequential data packets to the acceptor, some mechanisms allow the context acceptor to check whether or not the packets are arriving as they should: in the right order, and with no unwanted duplication of packets (shown in Figure 1–9). The acceptor checks for these two conditions when it verifies a packet's validity or when it unwraps a packet; see "Unwrapping and Verification" on page 59 for more information.

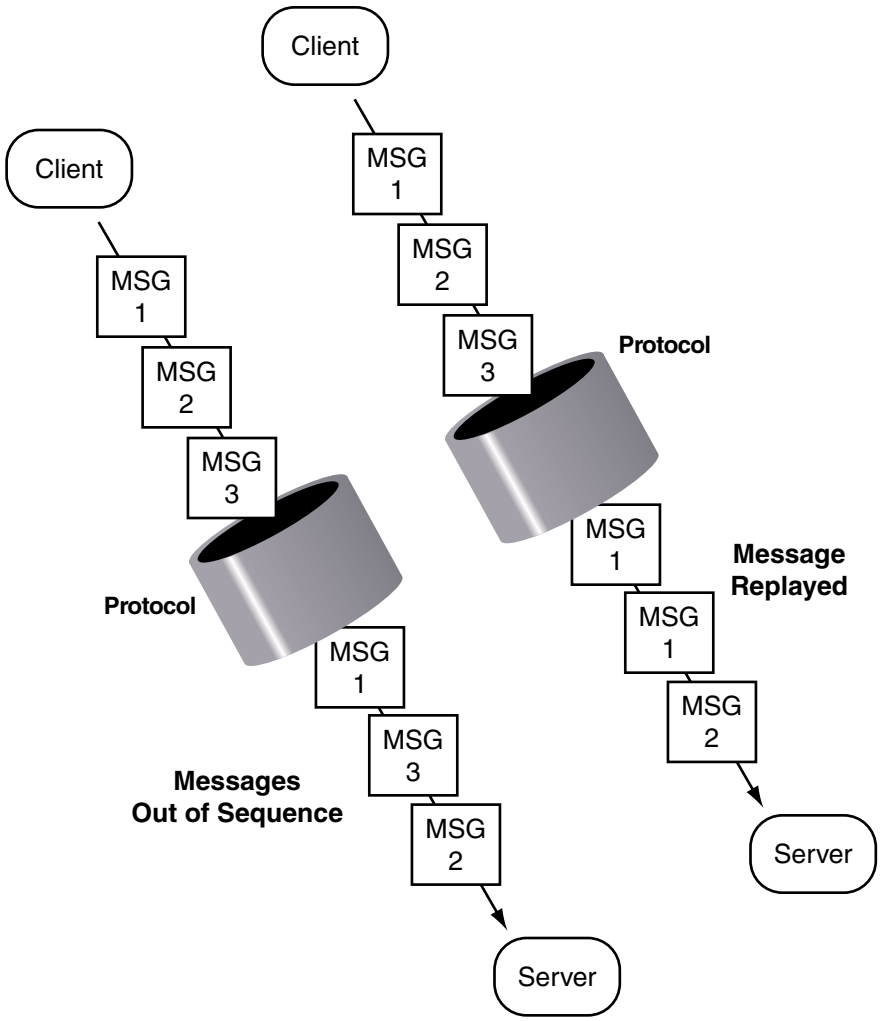

FIGURE 1–9 Message Replay and Message Out-of-Sequence

To request that these two conditions be looked for, the initiator should logically `OR` the *req_flags* argument with the values GSS_C_REPLAY_FLAG or GSS_C_SEQUENCE_FLAG when initiating the context with `gss_init_sec_context()`.

Anonymous Authentication

In normal use of the GSS-API, the initiator's identity is made available to the acceptor as a result of the context establishment process. However, context initiators can request that their identity not be revealed to the context acceptor.

As an example, consider an application providing access to a database containing medical information, and offering unrestricted access to the service. A client of such a service might want to authenticate the service (in order to establish trust in any information retrieved from it), but might not want the service to be able to obtain the client's identity (perhaps due to privacy concerns about the specific inquiries, or perhaps to avoid being placed on mailing lists).

To request anonymity, set the *req_flags* argument of `gss_init_sec_context()` to `GSS_C_ANON_FLAG`; to check if anonymity is available, check the *ret_flags* argument to `gss_init_sec_context()` or `gss_accept_sec_context()` to see if this same value is returned.

If anonymity is in effect and `gss_display_name()` is called on a client name returned by `gss_accept_sec_context()` or `gss_inquire_context()`, `gss_display_name()` will produce a generic anonymous name.

Note – It is the application's responsibility to take appropriate action if anonymity is requested but not permitted — the GSS-API will not terminate a context on these grounds.

Channel Bindings

For many applications, basic context establishment is sufficient to assure proper authentication of a context initiator. In cases where additional security is desired, the GSS-API offers the use of *channel bindings*. Channel bindings are tags that identify the particular data channel being used — that is, the origin and endpoint (initiator and acceptor) of the context. Because these tags are specific to the originator and recipient applications, they offer more proof of a valid identity.

Channel bindings are pointed to by the `gss_channel_bindings_t` data type, which is a pointer to a `gss_channel_bindings_struct` structure as shown in Example 1–7:

EXAMPLE 1–7 `gss_channel_bindings_t`

```
typedef struct gss_channel_bindings_struct {
OM_uint32       initiator_addrtype;
gss_buffer_desc initiator_address;
OM_uint32       acceptor_addrtype;
gss_buffer_desc acceptor_address;
gss_buffer_desc application_data;
```

EXAMPLE 1–7 `gss_channel_bindings_t` *(Continued)*

```
} *gss_channel_bindings_t;
```

The first two fields are the address of the initiator along with an address type that identifies the format in which the initiator's address is being sent. For example, the *inititiator_addrtype* might be sent to GSS_C_AF_INET to indicate that the *initiator_address* is in the form of an Internet address — that is, an IP address. Similarly, the third and fourth fields indicate the address and address type of the acceptor. The final field, *application_data*, can be used by the application as it wants (it's good programming practice to set it to GSS_C_NO_BUFFER if you're not going to use it). If an application does not want to specify an address, it should set its address type field to GSS_C_AF_NULLADDR. "Address Types for Channel Bindings" on page 124 has a list of valid address type values.

These address types indicate address families, rather than specific addressing formats. For address families that contain several alternative address forms, the *initiator_address* and *acceptor_address* fields must contain sufficient information to determine which address form is used. When not otherwise specified, addresses should be specified in network byte-order (that is, native byte-ordering for the address family).

To establish a context using channel bindings, the *input_chan_bindings* argument for `gss_init_sec_context()` should point to an allocated channel bindings structure. The function concatenates the structure's fields into an octet string, calculates a MIC over this string, and binds the MIC to the output token produced by `gss_init_sec_context()`. The application then sends the token to the context acceptor, which receives it and calls `gss_accept_sec_context()`. (See "Context Acceptance (Server)" on page 40.) `gss_accept_sec_context()` calculates a MIC on the received channel bindings and returns `GSS_C_BAD_BINDINGS` if the MIC does not match.

Because `gss_accept_sec_context()` returns the transmitted channel bindings, an acceptor can do its own security checking based on the received channel binding values. For example, it might check the value of *application_data* against code words kept in a secure database. However, in many cases this is "overkill."

Note – An underlying mechanism might or might not provide confidentiality for channel binding information. Therefore, an application should not include sensitive information as part of channel bindings unless it knows that confidentiality is ensured. The application might check the *ret_flags* argument of `gss_init_sec_context()` or `gss_accept_sec_context()`, especially for the values `GSS_C_CONF_FLAG` and `GSS_C_PROT_READY_FLAG` in order to determine if confidentiality is available. See "Context Initiation (Client)" on page 34 or "Context Acceptance (Server)" on page 40 for information on *ret_flags*.

Individual mechanisms can impose additional constraints on addresses and address types that can appear in channel bindings. For example, a mechanism can verify that the *initiator_address* field of the channel bindings presented to `gss_init_sec_context()` contains the correct network address of the host system. Portable applications should therefore ensure that they either provide correct information for the address fields or omit addressing information, specifying GSS_C_AF_NULLADDR as the address types.

Context Export and Import

The GSS-API provides a means for exporting and importing a context. The primary reason for this ability is to allow a multiprocess application (usually the context acceptor) to transfer a context from one process to another. For example, an acceptor might have one process that listens for context initiators and another that processes data sent in a context. ("`test_import_export_context()`" on page 106 shows how a context can be saved and restored with these functions.)

The function `gss_export_sec_context()` creates an interprocess token that contains information about the exported context. (See "Interprocess Tokens" on page 27. This buffer to receive the token should be set to GSS_C_NO_BUFFER before `gss_export_sec_context()` is called.)

The application then passes the token on to the other process, which accepts it and passes it to `gss_import_sec_context()`. The same functions used to pass tokens between applications can often be used to pass them between processes as well.

Only one instantiation of a security process can exist at a time. `gss_export_sec_context()` deactivates the exported context and sets its context handle to GSS_C_NO_CONTEXT. It also deallocates any and all process-wide resources associated with that context. In the event that context exportation cannot be completed, `gss_export_sec_context()` does not return an interprocess token, but leaves the existing security context unchanged.

Not all mechanisms permit contexts to be exported. An application can determine whether a context can be exported by checking the *ret_flags* argument to `gss_accept_sec_context()` or `gss_init_sec_context()`. If this flag is set to GSS_C_TRANS_FLAG, then the context can be exported. (See "Context Acceptance (Server)" on page 40 and "Context Initiation (Client)" on page 34.)

Figure 1–10 shows how a multiprocess acceptor might use context exporting to multitask. In this case, Process 1 receives and processes tokens, separating the context-level tokens from the data tokens, and passes the tokens on to Process 2, which deals with data in an application-specific way. In this illustration, the clients have already gotten export tokens from `gss_init_sec_context()`; they pass them to a user-defined function, `send_a_token()`, which indicates whether the token it's

transmitting is a context-level token or a message token. `send_a_token()` transmits the tokens to the server. Although not shown here, `send_a_token()` would presumably be used to pass tokens between threads as well.

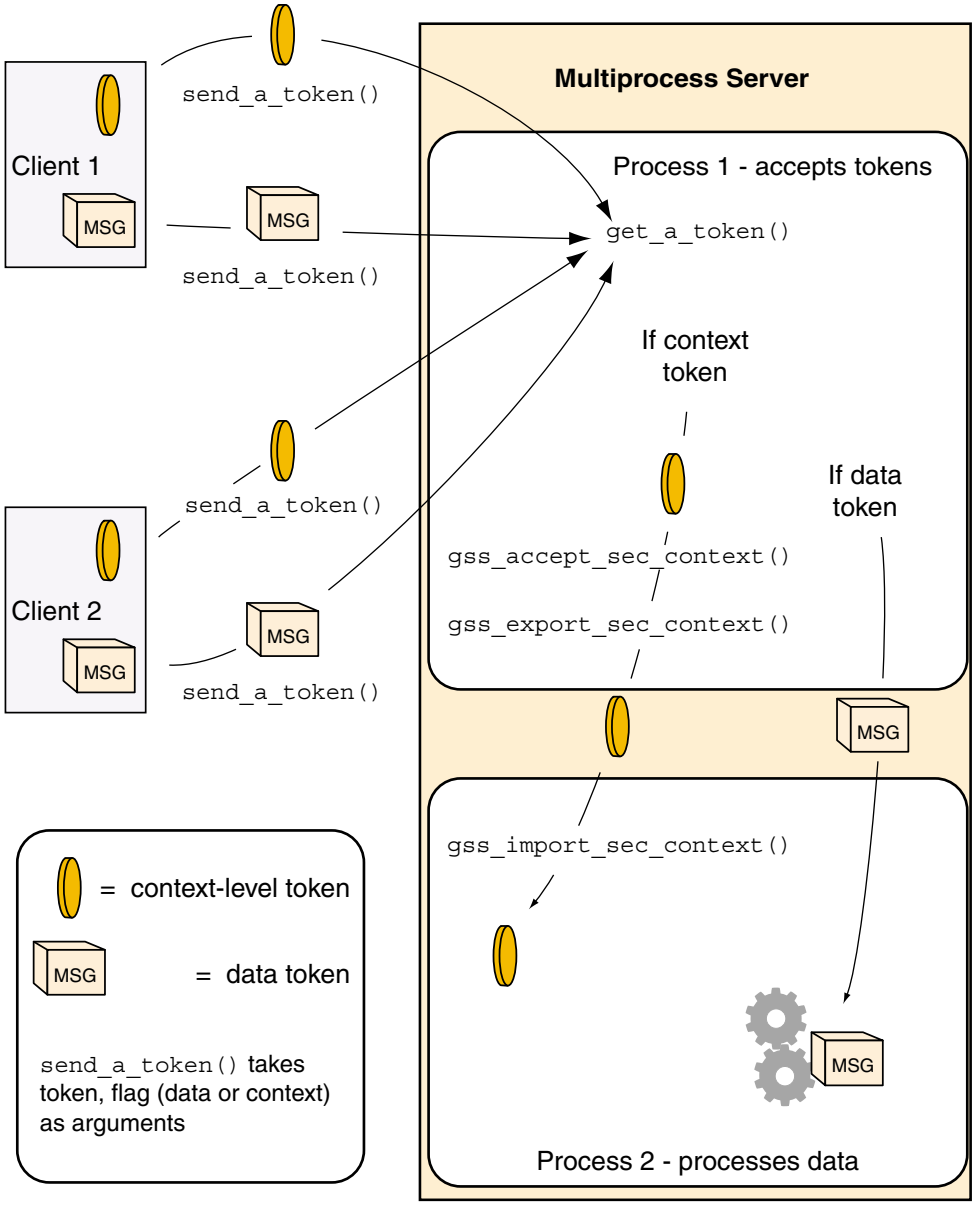

FIGURE 1–10 Exporting Contexts: Multithreaded Acceptor Example

Context Information

The GSS-API provides a function, `gss_inquire_context()`, that obtains information about a given security context (even an incomplete one). Given a context handle, `gss_inquire_context()` provides the following information about it:

- The name of the context initiator.
- The name of the context acceptor.
- The number of seconds for which the context will remain valid.
- The security mechanism used with the context.
- Several context-parameter flags. These flags are the same as the *ret_flags* argument of the `gss_accept_sec_context()` function (see "Context Acceptance (Server)" on page 40), covering delegation, mutual authentication, and so on.
- A flag indicating whether or not the inquiring application was the context initiator.
- A flag indicating whether or not the context is fully established.

For more information, see the `gss_inquire_context(3GSS)` man page.

Data Protection

After a context has been established between two peers — say, a client and a server — messages can be protected before being sent.

If you only establish a context and then send a message, you are utilizing the most basic GSS-API protection: *authentication*, wherein the recipient knows that the message comes from the principal claiming to be the sender. Depending on the underlying security mechanism being used, the GSS-API provides two other levels of protection:

- *Integrity* — The message is given a Mechanism Integrity Code (MIC) that can be checked by the recipient to ensure that the received message is the same as the one sent. The GSS-API function `gss_get_mic()` generates a MIC.
- *Confidentiality* — In addition to receiving a MIC, the message is encrypted. The GSS-API function `gss_wrap()` performs the encryption.

The difference between the two functions is shown in Figure 1–11.

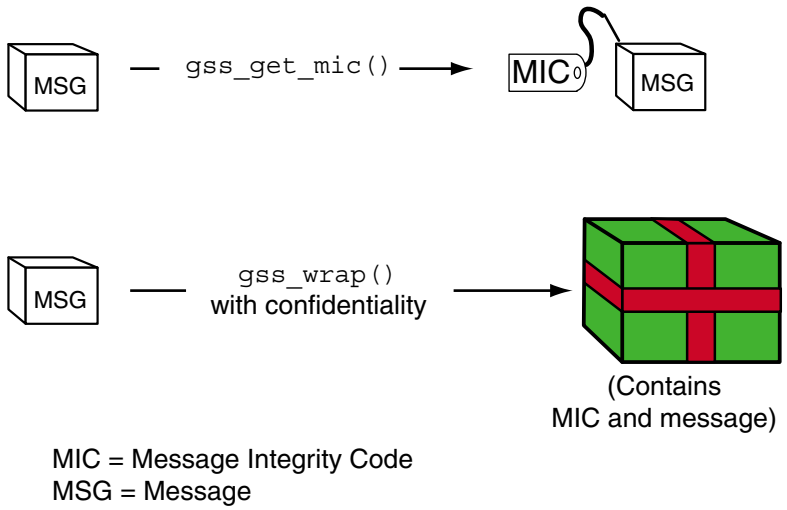

FIGURE 1–11 `gss_get_mic()` vs. `gss_wrap()`

Which function you use depends on your needs. Because `gss_wrap()` includes the integrity service, many programs use `gss_wrap()`. They can test for the availability of the confidentiality service, calling `gss_wrap()` with or without it, depending on whether it's available. An example is "Sending the Data" on page 71 (program listing in "`call_server()`" on page 85). However, since messages protected with `gss_get_mic()` don't need to be unwrapped by a recipient, there is a savings in CPU cycles over using `gss_wrap()`. Thus a program that doesn't need confidentiality may prefer to protect messages with `gss_get_mic()`.

Message Tagging With `gss_get_mic()`

Programs can use `gss_get_mic()` to add a cryptographic MIC to a message; the recipient can check this MIC to see if the received message is the same as the one that was sent by calling `gss_verify_mic()`. `gss_get_mic()` has the following form:

```
OM_uint32 gss_get_mic (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
gss_qop_t          qop_req,
const gss_buffer_t message_buffer,
gss_buffer_t       msg_token)
```

minor_status The status code returned by the underlying mechanism.

context_handle	The context under which the message will be sent.
qop_req	A requested QOP (Quality of Protection). This is the cryptographic algorithm used in generating the MIC. For portability's sake, applications should specify the default QOP by setting this argument to GSS_C_QOP_DEFAULT whenever possible. (See Appendix C on specifying a non-default QOP.)
message_buffer	The message to be tagged with a MIC. This argument must be in the form of a `gss_buffer_desc` object; see "Strings and Similar Data" on page 16. Must be freed up with `gss_release_buffer()` when you have finished with it.
msg_token	The token containing the message and its MIC. This must be freed up with `gss_release_buffer()` when you have finished with it.

Note that `gss_get_mic()` produces separate output for the message and the MIC. (This is different from `gss_wrap()`, which bundles them together as output.) This separation means that a sender application must arrange to send both the message and its MIC. More significantly, the receiving application must be able to receive and distinguish the message and the MIC. Ways to ensure the proper processing of message and MIC include:

- Through program control (that is, state). A recipient application might know to call its receiving function twice, once to get a message, once to get the message's MIC.
- Through flags. Sending and receiving functions can flag what kind of token they're including.
- Through user-defined token structures that might include both message and MIC.

`gss_get_mic()` returns `GSS_S_COMPLETE` if it completes successfully. If the specified QOP is not valid, it returns `GSS_S_BAD_QOP`. For more information, see the `gss_get_mic(3GSS)` man page.

Message Wrapping With `gss_wrap()`

Messages can also be "wrapped" by the `gss_wrap()` function. Like `gss_get_mic()`, `gss_wrap()` provides a MIC; it also encrypts a given message, if confidentiality is requested (and permitted by the underlying mechanism). The message receiver "unwraps" the message with `gss_unwrap()`. `gss_wrap()` looks like this:

```
OM_uint32 gss_wrap (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
int                conf_req_flag,
gss_qop_t          qop_req
const gss_buffer_t input_message_buffer,
int                *conf_state,
gss_buffer_t       output_message_buffer )
```

minor_status	The status code returned by the underlying security mechanism.
context_handle	The context under which this message will be sent.
conf_req_flag	A flag for requesting the confidentiality service (encryption). If non-zero, both confidentiality and integrity are requested; if zero, only the integrity service is requested.
qop_req	A requested QOP (Quality of Protection). This is the cryptographic algorithm used in generating the MIC and doing the encryption. For portability's sake, applications should specify the default QOP by setting this argument to GSS_C_QOP_DEFAULT whenever possible. (See Appendix C on specifying a non-default QOP.)
input_message_buffer	The message to be wrapped. This argument must be in the form of a `gss_buffer_desc` object; see "Strings and Similar Data" on page 16. Must be freed up with `gss_release_buffer()` when you have finished with it.
conf_state	A flag that, on the function's return, indicates whether confidentiality was applied or not. If non-zero, confidentiality, message origin authentication, and integrity services were applied. If zero, only message-origin authentication and integrity were applied. Specify NULL if not required.
output_message_buffer	The buffer for the wrapped message. After the application is done with the message, it must release this buffer with `gss_release_buffer()`.

Unlike `gss_get_mic()`, `gss_wrap()` wraps the message and its MIC together in the outgoing message, so the function that transmits them need be called only once. On the other end, `gss_unwrap()` will extract the message (the MIC is not visible to the application).

`gss_wrap()` returns `GSS_S_COMPLETE` if the message was successfully wrapped. If the requested QOP was not valid, it returns `GSS_S_BAD_QOP`. "Sending the Data" on page 71 (listing in "`call_server()`" on page 85) shows an example of `gss_wrap()` being used. For more information, see the `gss_wrap(3GSS)` man page.

Wrap Size

Wrapping a message with `gss_wrap()` increases its size. Because the protected message packet must not be too big to "fit through" a given transportation protocol, the GSS-API provides a function, `gss_wrap_size_limit()`, that calculates the

maximum size of a message that can be wrapped without becoming too large. The application can break up messages that exceed this size before calling `gss_wrap()`. It's a good idea to check the wrap-size limit before actually wrapping the message.

The amount of the size increase depends on two things:

- Which QOP (Quality of Protection) algorithm is used for making the transformation. Since the default QOP can vary from one implementation of the GSS-API to another, a wrapped message can vary in size even if you do not specify a non-default QOP. This is shown in the following figure.

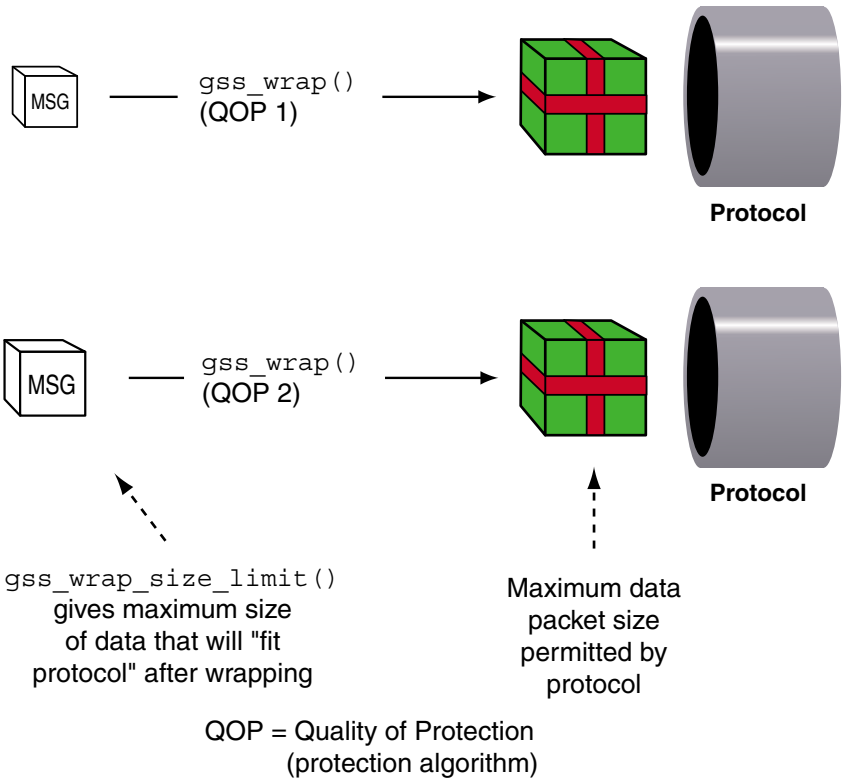

FIGURE 1–12 Wrap Size (Different QOPs)

- Whether confidentiality is invoked. Whether or not confidentiality is applied, `gss_wrap()` still increases the size of a message, because it embeds a MIC into the transmitted message. However, encrypting the message can further increase the size. The following figure shows how this works.

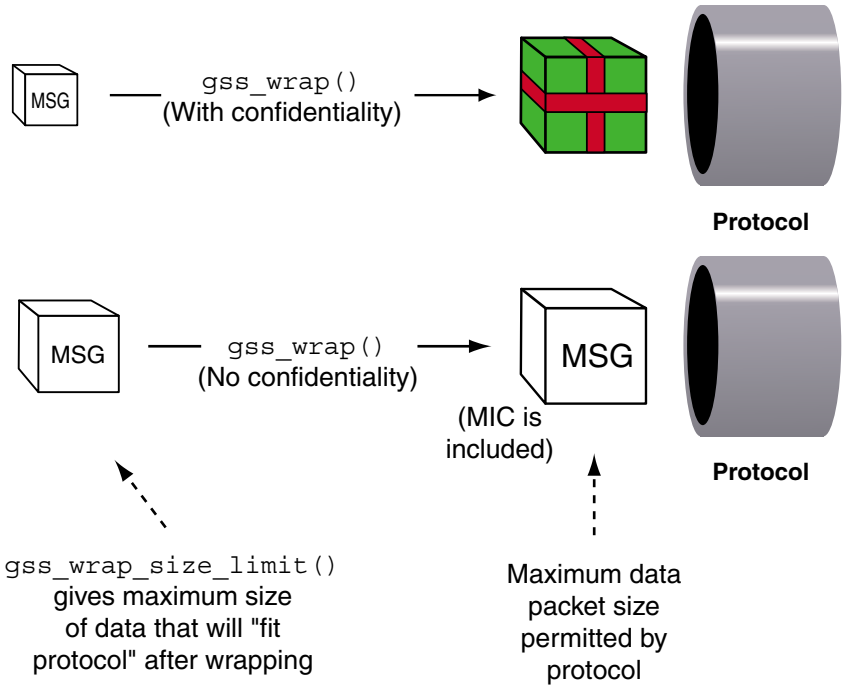

FIGURE 1–13 Wrap Size (Confidentiality/No Confidentiality)

`gss_wrap_size_limit()` looks like this:

```
OM_uint32 gss_wrap_size_limit (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
int                conf_req_flag,
gss_qop_t          qop_req,
OM_uint32          req_output_size,
OM_uint32          *max_input_size)
```

minor_status	The status code returned by the underlying mechanism.
context_handle	The context under which the data is transmitted.
conf_req_flag	A flag for requesting the confidentiality service (encryption). If non-zero, both confidentiality and integrity are requested; if zero, only the integrity service is requested.
qop_req	A requested QOP (Quality of Protection). This is the cryptographic algorithm used in generating the MIC and doing the encryption. For portability's sake, applications should specify the default QOP

by setting this argument to GSS_C_QOP_DEFAULT whenever possible. (See Appendix C on specifying a non-default QOP.)

req_output_size The maximum size (as an `int`) of a data chunk that a given transport protocol can handle. You must provide this information yourself; since the GSS-API is protocol-independent, it has no way of knowing which protocol is being used.

max_input_size Returned by the function, this is the maximum size of an unwrapped message that, when wrapped, will not exceed *req_output_size*.

`gss_wrap_size_limit()` returns `GSS_S_COMPLETE` if it completes successfully. If the specified QOP was not valid, it returns `GSS_S_BAD_QOP`. "`call_server()`" on page 85 includes an example of `gss_wrap_size_limit()` being used to return the maximum original message size, both if confidentiality is used and if it is not used.

Successful completion of this call does not necessarily guarantee that `gss_wrap()` will be able to protect a message of length *max_input_size* bytes, since this ability can depend on the availability of system resources at the time that `gss_wrap()` is called. For more information, see the `gss_wrap_size_limit(3GSS)` man page.

Unwrapping and Verification

Once it has been received, a wrapped message must be unwrapped with `gss_unwrap()`. `gss_unwrap()` automatically verifies the message against the MIC that is embedded with the wrapped message. If the sender did not wrap the message but used `gss_get_mic()` to produce a MIC, then the received message can be verified against that MIC with `gss_verify_mic()`. In this latter case the acceptor must arrange to receive both the message and its MIC.

`gss_unwrap()`

`gss_unwrap()` looks like this:

```
OM_uint32 gss_unwrap (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
const gss_buffer_t input_message_buffer,
gss_buffer_t       output_message_buffer,
int                *conf_state
gss_qop_t          *qop_state)
```

minor_status The status code returned by the underlying security mechanism.

context_handle	The context under which this message will be sent.
input_message_buffer	The wrapped message. This argument must be in the form of a `gss_buffer_desc` object; see "Strings and Similar Data" on page 16. Must be freed up with `gss_release_buffer()` when you have finished with it.
output_message_buffer	The buffer for the unwrapped wrapped message. After the application is done with the unwrapped message, it must release this buffer with `gss_release_buffer()`. This argument is also a `gss_buffer_desc` object.
conf_state	A flag that indicates whether confidentiality was applied or not. If non-zero, then confidentiality, message origin authentication, and integrity services were applied. If zero, only message-origin authentication and integrity were applied. Specify NULL if not required.
qop_state	The QOP (Quality of Protection) used. This is the cryptographic algorithm used in generating the MIC and doing the encryption. Specify NULL if not required.

`gss_unwrap()` returns `GSS_S_COMPLETE` if the message was successfully unwrapped. If it cannot verify the message against its MIC, it returns `GSS_S_BAD_SIG`.

gss_verify_mic()

If a message has been unwrapped, or if it was never wrapped in the first place, it can be verified with `gss_verify_mic()`. `gss_verify_mic()` looks like this:

```
OM_uint32 gss_verify_mic (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
const gss_buffer_t message_buffer,
const gss_buffer_t token_buffer,
gss_qop_t          qop_state)
```

minor_status	The status code returned by the underlying mechanism.
context_handle	The context under which the message will be sent.
message_buffer	The received message. This argument must be in the form of a `gss_buffer_desc` object; see "Strings and Similar Data" on page 16. Must be freed up with `gss_release_buffer()` when you have finished with it.
token_buffer	The token containing the received MIC. This argument must be in the form of a `gss_buffer_desc` object; see "Strings and Similar

Data" on page 16. This buffer must be freed up with `gss_release_buffer()` when the application has finished with it.

qop_state — The QOP (Quality of Protection) that was applied in generating the MIC. Specify NULL if not required.

`gss_verify_mic()` returns `GSS_S_COMPLETE` if the message was successfully verified. If it cannot verify the message against its MIC, it returns `GSS_S_BAD_SIG`.

Transmission Confirmation (Optional)

After the recipient has unwrapped or verified the transmitted message, it might want to send a confirmation to the sender. This means sending back a MIC for that message. Consider the case of a message that was not wrapped by the sender, but only tagged with a MIC with `gss_get_mic()`. The process, illustrated in Figure 1–14, is as follows:

1. The initiator tags the message with `gss_get_mic()`.
2. The initiator sends the message and MIC to the acceptor.
3. The acceptor verifies the message with `gss_verify_mic()`.
4. The acceptor sends the MIC back to the initiator.
5. The initiator verifies the received MIC against the original message with `gss_verify_mic()`.

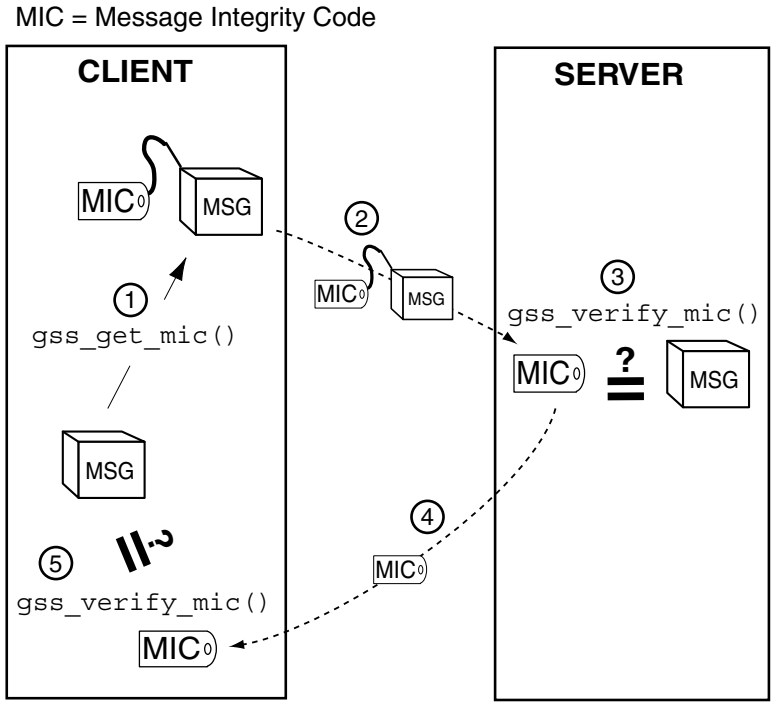

FIGURE 1–14 Confirming MIC'd Data

In the case of wrapped data, the `gss_unwrap()` function never produces a separate MIC, so the recipient must generate it from the received (and unwrapped) message. The process, illustrated in Figure 1–15, is as follows:

1. The initiator wraps the message with `gss_wrap()`.
2. The initiator sends the wrapped message.
3. The acceptor unwraps the message with `gss_unwrap()`.
4. The acceptor calls `gss_get_mic()` to produce a MIC for the unwrapped message.
5. The acceptor sends the derived MIC to the initiator.
6. The initiator compares the received MIC against the original message with `gss_verify_mic()`.

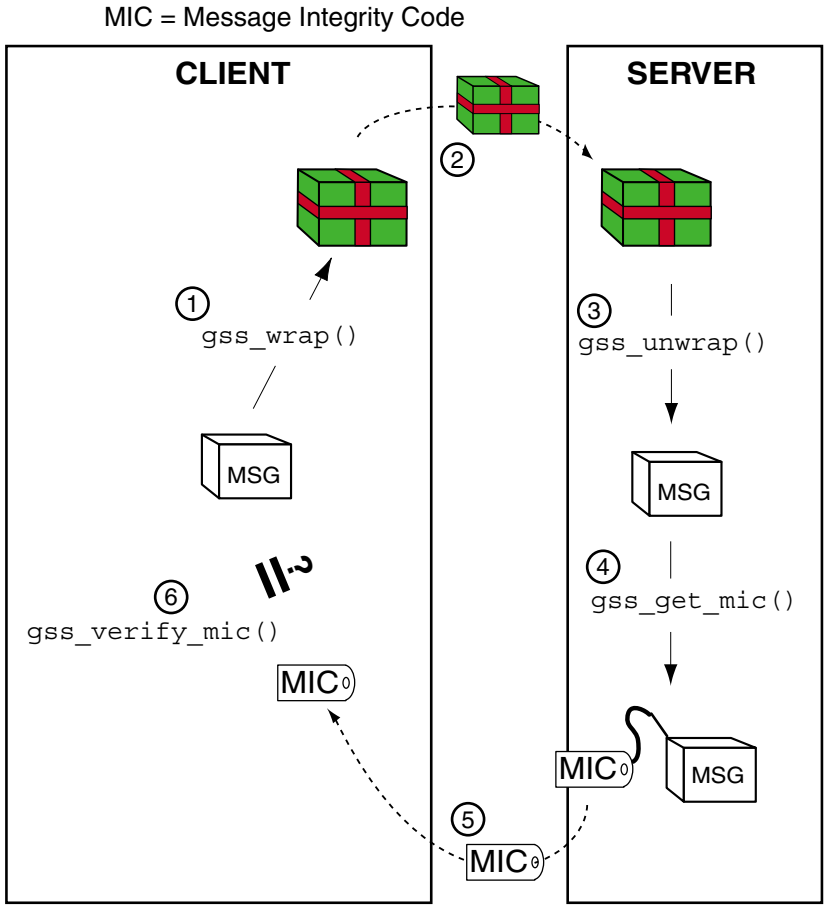

FIGURE 1–15 Confirming Wrapped Data

Context Deletion and Data Deallocation

After all messages have been sent and received, and the initiator and acceptor applications have finished, *both applications* should call `gss_delete_sec_context()` to destroy their shared context. `gss_delete_sec_context()` deletes local data structures associated with the context. `gss_delete_sec_context()` looks like this:

```
OM_uint32 gss_delete_sec_context (
OM_uint32    *minor_status,
gss_ctx_id_t *context_handle,
```

```
gss_buffer_t output_token)
```

minor_status	The status code returned by the underlying mechanism.
context_handle	The context to delete.
output_token	Should be set to GSS_C_NO_BUFFER.

See the `gss_delete_sec_context(3GSS)` man page for more information.

For good measure, applications should be sure to deallocate any data space they have allocated for GSS-API data. The functions that do this are `gss_release_buffer()`, `gss_release_cred()`, `gss_release_name()`, and `gss_release_oid_set()`. See their man pages for more information.

CHAPTER 2

A Walk–Through of the Sample GSS-API Programs

This chapter describes two sample GSS-API application programs. This chapter provides the following information:

Introduction to the Sample Programs

The source code for two C-language applications that make use of the GSS-API is provided in Appendix A. One sample application is for a client and the other for a server. This chapter guides you through those applications, step-by-step; it is intended to be read while referring to the cource code. It does not attempt to explain every facet of the applications in detail. Rather, it focuses on the aspects that relate to using the GSS-API.

Caution – Because the GSS_API does not automatically clean up after itself, applications and functions using the GSS-API must do so themselves. This means that functions that use GSS-API buffers or GSS-API namespaces, for example, should call GSS-API functions such as `gss_release_buffer()` and `gss_release_name()` when they are finished.

To save space and avoid repetition, we have generally not included such cleanup in the following code walk-through. Be aware that it must be performed. However; refer to Appendix A to see the sample programs in full if you are unsure how or when to use the cleanup functions.

Client-Side GSS-API: gss-client

The sample client-side program, gss-client, creates a security context with a server, establishes security parameters, and sends a string (the "message") to the server. It uses a simple TCP-based sockets connection to make its connection.

gss-client takes this form on the command line:

```
gss-client [-port port] [-d] [-mech mech] host service [-f] msg
```

Specifically, `gss-client` does the following:

1. Parses the command line.
2. Creates an OID (object ID) for a mechanism, if specified.
3. Creates a connection to the server.
4. Establishes a context.
5. Wraps the message.
6. Sends the message.
7. Verifies that the message has been "signed" correctly by the server.

Following is a step-by-step description of how gss-client works. Because it is a sample program designed to show off functionality, the parts of the program that do not closely relate to the steps above are skipped. Some features, such as importing and exporting contexts, or getting a wrap size, are discussed elsewhere in this manual.

Overview: `main()` (Client)

As with all C programs, the outer shell of the program is contained in the entry-point function, `main()`. `main()` performs four functions:

1. It parses command-line arguments, assigning them to variables:
   - If specified, *port* is the port number for making the connection to the remote machine specified by *host*.
   - If the `-d` flag is set, security credentials should be delegated to the server. Specifically, the *deleg_flag* variable is set to the GSS-API value `GSS_C_DELEG_FLAG`; otherwise *deleg_flag* is set to zero.
   - *mech* is the (optional) name of the security mechanism, such as Kerberos v5 or X.509, to use. If no mechanism is specified, the GSS-API will use a default mechanism.
   - The name of the network service requested by the client (such as telnet, ftp, or login service) is assigned to *service_name*.

- Finally, *msg* is the string to send to the server as protected data. If the `-f` option is specified, then *msg* is the name of a file from which to read the string.

An example command line might look like this:

```
% gss-client -port 8080 -d -mech kerberos_v5 erebos.eng nfs "ls"
```

This command line specifies neither mechanism nor port, and does not use delegation:

```
% gss-client erebos.eng nfs "ls"
```

2. It calls `parse_oid()` to create a GSS-API OID (object identifier) from the name of a security mechanism (if such a name has been provided on the command line):

```
if (mechanism)
        parse_oid(mechanism, &g_mechOid);
```

where *mechanism* is the string to translate and *g_mechOid* is a pointer to a `gss_OID` object for the mechanism. See Appendix C for more about specifying a non-default mechanism.

3. It calls `call_server()`, which does the actual work of creating a context and sending data.

```
if (call_server(hostname, port, g_mechOid, service_name,
                    deleg_flag, msg, use_file) < 0)
          exit(1);
```

4. It releases the storage space for the OID if it has not been released yet.

```
if (g_mechOID != GSS_C_NULL_OID)
     (void) gss_release_oid(&min_stat, &g_mechoid);
```

Note that `gss_release_oid()`, while supported by the Sun implementation of the GSS-API, is not supported by all GSS-API implementations and is considered nonstandard. Since applications should if possible use the default mechanism provided by the GSS-API instead of allocating one (with `gss_str_to_oid()`), the `gss_release_oid()` command generally should not be used.

Specifying a Non-Default Mechanism

As a general rule, any application using the GSS-API should not attempt to specify a particular mechanism, but instead use the default mechanism provided by the GSS-API implementation. The default mechanism is specified by setting the `gss_OID` representing the mechanism to the value `GSS_C_NULL_OID`.

Because setting a non-default mechanism is not recommended, this part of the program does not cover it here. Interested readers can see how the client application parses the user-input mechanism name by looking at the code in "`parse_oid()`" on page 84 and by looking at Appendix C, which explains how to using non-default OIDs.

Calling the Server

After the mechanism has been put in the form of a gss_OID, you can do the actual work, so `main()` now calls the function `call_server()` with much the same arguments as on the command line.

```
call_server(hostname, port, g_mechOid, service_name,
     deleg_flag, msg, use_file);
```

(*use_file* is a flag indicating whether the message to be sent is contained in a file or not.)

Connecting to the Server

After declaring its variables, `call_server()` first makes a connection with the server:

```
if ((s = connect_to_server(host, port)) < 0)
     return -1;
```

where *s* is a file descriptor (an `int`, initially returned by a call to `socket()`).

`connect_to_server()` is a simple function that uses sockets to create a connection. Because it doesn't use the GSS-API, it's skipped here. You can see it at "`connect_to_server()`" on page 94.

Establishing a Context

After the connection is established, `call_server()` uses the function `client_establish_context()` to, yes, establish the security context:

```
int client_establish_context(s, service_name, deleg_flag, oid,
     &context, &ret_flags)
```

where

- *s* is a file descriptor representing the connection established by `connect_to_server()`.
- *service_name* is the network service requested (for example, `nfs`).
- *deleg_flag* specifies whether or not the server may act as a proxy for the client.
- *oid* is the mechanism.
- *context* is the context to be created.
- *ret_flags* is an int that specifies any flags returned by the GSS-API function `gss_init_sec_context()`.

To initiate the context, the application uses the function `gss_init_sec_context()`. As this function, like most GSS-API functions, requires names to be in internal GSS-API format, the application must first translate the service name from a string to internal format. For that, it can use `gss_import_name()`:

```
maj_stat = gss_import_name(&min_stat, &send_tok,
     (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &target_name);
```

This function takes the name of the service (stored in an opaque GSS_API buffer, `send_tok`) and converts it to the GSS_API internal name `target_name`. (`send_tok` is used to save space, instead of declaring a new `gss_buffer_desc`.) The third argument is a `gss_OID` type that indicates the name format that `send_tok` has. In this case, it is `GSS_C_NT_HOSTBASED_SERVICE`, which means a service of the format *service@host*. (See "Name Types" on page 123 for other possible values for this argument.)

Once the service has been rendered in GSS-API internal format, we can proceed with establishing the context. In order to maximize portability, context-establishment should always be performed as a loop (see "Context Initiation (Client)" on page 34).

First, the application initializes the context to be null:

`*gss_context = GSS_C_NO_CONTEXT;`It does the same for the token that we'll receive from the server:

```
token_ptr = GSS_C_NO_BUFFER;
```

The application now enters the loop. The loop proceeds by checking two things: the status returned by `gss_init_sec_context()` and the size of the token to be sent to the server (also generated by `gss_init_sec_context()`). If the token's size is zero, then the server is not expecting another token from the client. The pseudocode for the loop that follows looks like this:

```
do
   gss_init_sec_context()
   if no context was created
      uh-oh.  Exit with error;
   if the status is neither "complete" nor "in process"
      uh-oh.  Release the service namespace and exit with error;
   if there's a token to send to the server (= if its size is nonzero)
      send it;
      if sending it fails,
         oops!  release the token and the service
            namespaces and exit with error;
      release the namespace for the token we've just sent;
   if we're not done setting up the context
      receive a token from the server;
while the context is not complete
```

First, the call to `gss_init_sec_context()`:

```
do {
     maj_stat = gss_init_sec_context(&min_stat,
                           GSS_C_NO_CREDENTIAL,
                           gss_context,
```

```
target_name
oid
GSS_C_MUTUAL_FLAG |
   GSS_C_REPLAY_FLAG |
   deleg_flag,
0,
NULL,
&send_tok,
ret_flags,
NULL);
```

where the arguments are as follows:

- The status code to be set by the underlying mechanism.
- The credential handle. We use `GSS_C_NO_CREDENTIAL` to act as a default principal.
- (*gss_context*) The context handle to be created.
- (*target_name*) The service, as a GSS_API internal name.
- (*oid*) The mechanism.
- Request flags. In this case, the client requests that a) the server authenticate itself, b) that message-duplication be turned on, and c) that the server act as a proxy if requested.
- No time limit for the context.
- No request for channel bindings.
- (*token_ptr*) Pointer to the token received from the server, if any.
- The mechanism actually used by the server (set to `NULL` here because the application isn't interested in this value).
- (*&send_tok*) The token created by `gss_init_sec_context()` to send to the server.
- Return flags. Set to `NULL` because we ignore them.

You might have noticed that the client does not need to acquire credentials before initiating a context. On the client side, credential management is handled transparently by the GSS-API. That is, the GSS-API "knows" how to get credentials created by this mechanism for this principal (usually at login time). That is why the application passes `gss_init_sec_context()` a default credential. On the server side, however, a server application must explicitly acquire credentials for a service before accepting a context. See "Acquiring Credentials" on page 75.

After checking that it has a context (but not necessarily a complete one) and that `gss_init_sec_context()` is returning valid status, the application sees if `gss_init_sec_context()` has given it a token to send to the server. If it hasn't, it's because the server has indicated that it doesn't need (another) one. If it has, then send it to the server. If sending it fails, release the namespaces for it and the service, and exit. Remember, you can check for the presence of a token by looking at its length:

```
if (send_tok_length != 0) {
               if (send_token(s, &send_tok) < 0) {
                    (void) gss_release_buffer(&min_stat, &send_tok);
                    (void) gss_release_name(&min_stat, &target_name);
                    return -1;
               }
          }
```

`send_token()` is not a GSS-API function; it is a basic write-to-file function written by the user. (You can see it at "`send_token()`" on page 112.) *Note that the GSS-API does not send or receive tokens itself. It is the responsibility of the calling applications to send and receive any tokens created by the GSS-API.*

If the server doesn't have any (more) tokens to send, then `gss_init_sec_context()` returns `GSS_S_COMPLETE`. So if `gss_init_sec_context()` hasn't returned this value, the application knows there's another token out there to fetch. If the fetch fails it releases the service namespace and quit:

```
if (maj_stat == GSS_S_CONTINUE_NEEDED) {
               if (recv_token(s, &recv_tok) < 0) {
                    (void) gss_release_name(&min_stat, &target_name);
                    return -1;
```

Finally, the program resets its token pointers, and continues the loop until the context is completely established. Thus its `do` loop ends as follows:

```
} while (maj_stat == GSS_S_CONTINUE_NEEDED);
```

Sending the Data

Having established the security context, gss-client needs to wrap the data, send it, and then verify the "signature" that the server returns. Because gss-client is an example program, it does various other things as well, such as display information about the context, but we'll skip all of that in order to get the data sent out and verified. So first the program puts the message to be sent (such as "ls") into a buffer:

```
if (use_file) {
    read_file(msg, &in_buf);
} else {
    /* Wrap the message */
    in_buf.value = msg;
    in_buf.length = strlen(msg) + 1;
}
```

Before wrapping, the program checks to see if it can encrypt the data:

```
if (ret_flag & GSS_C_CONF_FLAG) {
     state = 1;
else
```

```
        state = 0;
    }
```

And then it wraps it up:

```
    maj_stat = gss_wrap(&min_stat, context, conf_req_flag,
                        GSS_C_QOP_DEFAULT, &in_buf, &state, &out_buf);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("wrapping message", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context,
                        GSS_C_NO_BUFFER);
        return -1;
    } else if (! state) {
        fprintf(stderr, "Warning!  Message not encrypted.\n");
    }
```

Thus the message stored in *in_buf* is to be sent to the server referenced by *context*, with confidentiality service and the default Quality of Protection (QOP) requested. (Quality of Protection indicates which algorithm to apply in transforming the data; it's a good idea for portability's sake to use the default whenever possible.) `gss_wrap()` wraps the message, puts the result into *out_buf*, and sets a flag (*state*) that indicates whether confidentiality was in fact applied in the wrapping.

The client sends the wrapped message to the server with its own `send_token()` function, which you've already seen in "Establishing a Context" on page 68:

```
send_token(s, &outbuf)
```

Verifying the Message

The program can now verify the validity of the message it sent. It knows that the server returns the MIC for the message it sent, so it retrieves it with its `recv_token()` function and then uses `gss_verify_mic()` to verify its "signature" (the MIC).

```
    maj_stat = gss_verify_mic(&min_stat, context, &in_buf,
                              &out_buf, &qop_state);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("verifying signature", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context,
                                      GSS_C_NO_BUFFER);
        return -1;
    }
```

`gss_verify_mic()` compares the MIC received with the server's token (in *out_buf*) with one it produces from the original, unwrapped message, held in *in_buf*. If the two MICs match, the message is verified. The client releases the buffer for the received token, *out_buf*.

To finish, `call_server()` deletes the context and returns to `main()`.

Server-Side GSS-API: gss-server

Naturally, the client needs a server to perform a security handshake. Where the client initiates a security context and sends data, the server must accept the context, verifying the identity of the client. In doing so, it might need to authenticate itself to the client, if requested to do so, and it may have to provide a "signature" for the data to the client. Plus, of course, it has to process the data!

`gss-server` takes this form on the command line (the line has been broken up to make it fit):

```
gss-server [-port port] [-verbose] [-inetd] [-once] [-logfile file] \
                [-mech mechanism] service_name
```

`gss-server` does the following:

1. Parses the command line.
2. Translates the mechanism name given on the command-line, if any, to internal format.
3. Acquires credentials for the caller.
4. Checks to see if the user has specified using the inetd daemon for connecting or not.
5. Establishes a connection.
6. Gets the data.
7. Signs the data and returns it.
8. Releases namespaces and exits.

Following is a step-by-step description of how `gss-server` works. Because it is a sample program designed to show off functionality, the parts of the program that do not closely relate to the steps above are skipped here.

Overview: `main()` (Server)

gss-client begins with the `main()` function. `main()` performs the following tasks:

1. It parses command-line arguments, assigning them to variables:
   - *port*, if specified, is the port number to listen on. If no port is specified, the program uses port 4444 as the default.

- If `-verbose` is specified, the program runs in a quasi-debug mode.
- The `-inetd` option indicates that the program should use the `inetd` daemon to listen to a port; `inetd` uses `stdin` and `stdout` to hand the connection to the client.
- If `-once` is specified, then the program makes only a single-instance connection.
- *mechanism* is the (optional) name of the security mechanism to use, such as Kerberos v5, to use. If no mechanism is specified, the GSS-API uses a default mechanism.
- The name of the network service requested by the client (such as telnet, ftp, or login service) is specified by *service_name*.

An example command line might look like this:

```
% gss-server -port 8080 -once -mech kerberos_v5 erebos.eng nfs "hello"
```

2. It converts the mechanism, if specified, to a GSS-API object identifier (OID). This is because GSS-API functions handle names in internal format.
3. It acquires the credentials for the service (such as `ftp`), for the mechanism being used (for example, Kerberos v5).
4. It calls the `sign_server()` function, which does most of the work (establishes the connection, retrieves and signs the message, and so on).

   If the user has specified using `inetd`, then the program closes the standard output and standard error and calls `sign_server()` on the standard input, which `inetd` uses to pass connections. Otherwise, it creates a socket, accepts the connection for that socket with the TCP function `accept()`, and calls `sign_server()` on the file descriptor returned by `accept()`.

   If `inetd` is not used, the program creates connections and contexts until it's terminated. However, if the user has specified the `-once` option, the loop terminates after the first connection.
5. It releases the credentials it has acquired.
6. It releases the mechanism OID namespace.
7. It closes the connection, if it's still open.

Creating an OID for the Mechanism

As with the gss-client program example, the sample server program allows the user to specify a mechanism. However, it is strongly recommended that all applications use the default mechanism provided by the GSS-API implementation. The default mechanism is obtained by setting the `gss_OID` that represents the mechanism to `GSS_C_NULL_OID`. Interested readers can refer to the code itself in "`createMechOid()`" on page 99 and read about using non-default mechanisms in Appendix C.

Acquiring Credentials

As with the client application, neither the server application nor the GSS-API create credentials; they are created by the underlying mechanism(s). Unlike the client program, the server needs to explicitly acquire the credentials it needs. (Some client applications might want to acquire credentials explicitly, in which case they do so in the same manner as shown here. But generally the client has acquired credentials before that, at login time, and GSS-API acquires those automatically.)

The gss-server program has its own function, `server_acquire_creds()`, to get the credentials for the service being provided. It takes as its input the name of the service, and the security mechanism being used, then returns the credentials for the service.

`server_acquire_creds()` uses the GSS-API function `gss_acquire_cred()` to get the credentials for the service that the server provides. Before it can do this, however, it must do two things.

If a single credential can be shared by multiple mechanisms, `gss_acquire_cred()` returns credentials for all those mechanisms. Therefore, it takes as input not a single mechanism, but a *set* of mechanisms. (See "Credentials" on page 31.) However, in most cases, including this one, a single credential might not work for multiple mechanisms. Besides, in the server application, either a single mechanism is specified on the command line or the default mechanism is used. Therefore, the first thing to do is make sure that the set of mechanisms passed to `gss_acquire_cred()` contains a single mechanism, default or otherwise:

```
if (mechOid != GSS_C_NULL_OID) {
     desiredMechs = &mechOidSet;
     mechOidSet.count = 1;
     mechOidSet.elements = mechOid;
} else
     desiredMechs = GSS_C_NULL_OID_SET;
```

`GSS_C_NULL_OID_SET` indicates that the default mechanism should be used.

Because `gss_acquire_cred()` takes the service name in the form of a `gss_name_t` structure, the second thing to do is import the name of the service into that format. To do this, use `gss_import_name()`. Because this function, like all GSS-API functions, requires arguments to be GSS-API types, the service name has to be copied to a GSS-API buffer first:

```
     name_buf.value = service_name;
     name_buf.length = strlen(name_buf.value) + 1;
     maj_stat = gss_import_name(&min_stat, &name_buf,
                (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &server_name);
     if (maj_stat != GSS_S_COMPLETE) {
          display_status("importing name", maj_stat, min_stat);
          if (mechOid != GSS_C_NO_OID)
                gss_release_oid(&min_stat, &mechOid);
          return -1;
     }
```

Note again the use of the nonstandard function `gss_release_oid()`. See "Overview: `main()` (Client)" on page 66.

The input is the service name, as a string in *name_buf*, and the output is the pointer to a `gss_name_t` structure, *server_name*. The third argument, `GSS_C_NT_HOSTBASED_SERVICE`, is the name type for the string in *name_buf*; in this case it indicates that the string should be interpreted as a service of the format *service@host*.

Now the server program can call `gss_acquire_cred()`:

```
maj_stat = gss_acquire_cred(&min_stat, server_name, 0,
                                desiredMechs, GSS_C_ACCEPT,
                                server_creds, NULL, NULL);
```

Where:

- *min_stat* is the error code returned by the function.
- *server_name* is, as explained above, the name of the server.
- 0 indicates that the program isn't interested the maximum lifetime of the credential.
- *desiredMechs* is, as explained above, the set of mechanisms for which this credential applies.
- `GSS_C_ACCEPT` means that the credential can be used only to accept security contexts.
- *server_creds* is the credential handle to be returned by the function.
- NULL, NULL indicates that the program is not interested in knowing either the specific mechanism being employed nor the amount of time the credential will be valid.

Accepting a Context, Getting and Signing Data

Having acquired credentials for the service, the server program checks to see if the user has specified using inetd (see "Overview: `main()` (Server)" on page 73) and then calls `sign_server()`, which does the main work of the program. The first thing that `sign_server()` does is establish the context by calling `server_establish_context()`.

Note – inetd is not covered here. Basically, if inetd has been specified, the program calls `sign_server()` on the standard input. If not, it creates a socket, accepts a connection, and then calls `sign_server()` on that connection.

`sign_server()` does the following:

1. Accepts the context.
2. Unwraps the data.
3. Signs the data.
4. Returns the data.

Accepting a Context

Because establishing a context can involve a series of token exchanges between the client and the server, both context acceptance and context initialization should be performed in loops, to maintain program portability. Indeed, the loop for accepting a context is very similar to that for establishing one, although rather in reverse. (Compare with "Establishing a Context" on page 68.)

1. The first thing the server does is look for a token that the client should have sent as part of the context initialization process. Remember, the GSS-API does not send or receive tokens itself, so programs must have their own routines for performing these tasks. The one the server uses for receiving the token is called `recv_token()` (it can be found at "`recv_token()`" on page 113):

```
do {
     if (recv_token(s, &recv_tok) < 0)
          return -1;
```

2. Next, the program calls the GSS-API function `gss_accept_sec_context()`:

```
maj_stat = gss_accept_sec_context(&min_stat,
                                  context,
                                  server_creds,
                                  &recv_tok,
                                  GSS_C_NO_CHANNEL_BINDINGS,
                                  &client,
                                  &doid,
                                  &send_tok,
                                  ret_flags,
                                  NULL,     /* ignore time_rec */
                                  NULL);    /* ignore del_cred_handle */
```

where

- *min_stat* is the error status returned by the underlying mechanism.
- *context* is the context being established.
- *server_creds* is the credential for the service being provided (see "Acquiring Credentials" on page 75).
- *recv_tok* is the token received from the client by `recv_token()`.
- `GSS_C_NO_CHANNEL_BINDINGS` is a flag indicating not to use channel bindings (see "Channel Bindings" on page 49).
- *client* is the ASCII name of the client.

- *oid* is the mechanism (in OID format).
- *send_tok* is the token to send to the client.
- *ret_flags* are various flags indicating whether the context supports a given option, such as message-sequence-detection.
- NULL and NULL indicate that the program is not interested in the length of time the context will be valid, nor in whether the server can act as a client's proxy.

The acceptance loop continues (barring an error) as long as `gss_accept_sec_context()` sets *maj_stat* to `GSS_S_CONTINUE_NEEDED`. If *maj_stat* is not equal to either that value nor to `GSS_S_COMPLETE`, there's a problem and the loop exits.

3. `gss_accept_sec_context()` returns a positive value for the length of *send_tok* if there is a token to send back to the client. The next step is to see if there's a token to send, and, if so, to send it:

```
if (send_tok.length != 0) {
     . . .
     if (send_token(s, &send_tok) < 0) {
          fprintf(log, "failure sending token\n");
          return -1;
     }

     (void) gss_release_buffer(&min_stat, &send_tok);
     }
```

Unwrapping the Message

After accepting the context, the server receives the message sent by the client. Because the GSS-API doesn't provide a function to do this, the program uses its own function, `recv_token()`:

```
if (recv_token(s, &xmit_buf) < 0)
     return(-1);
```

Since the message might be encrypted, the program uses the GSS-API function `gss_unwrap()` to unwrap it:

```
maj_stat = gss_unwrap(&min_stat, context, &xmit_buf, &msg_buf,
                          &conf_state, (gss_qop_t *) NULL);
     if (maj_stat != GSS_S_COMPLETE) {
        display_status("unwrapping message", maj_stat, min_stat);
        return(-1);
     } else if (! conf_state) {
        fprintf(stderr, "Warning!  Message not encrypted.\n");
     }

     (void) gss_release_buffer(&min_stat, &xmit_buf);
```

`gss_unwrap()` takes the message that `recv_token()` has placed in *xmit_buf*, translates it, and puts the result in *msg_buf*. Two arguments to `gss_unwrap()` are noteworthy: *conf_state* is a flag to indicate whether confidentiality was applied for this message (that is, if the data is encrypted or not), and the final NULL indicates that the program isn't interested in the QOP used to protect the message.

Signing the Message, Sending It Back

All that is left, then, is for the server to "sign" the message — that is, to return the message's MIC (Message Integrity Code, a unique tag associated with message) to the client to prove that the message was sent and unwrapped successfully. To do that, the program uses the function `gss_get_mic()`:

```
maj_stat = gss_get_mic(&min_stat, context, GSS_C_QOP_DEFAULT,
                           &msg_buf, &xmit_buf);
```

which looks at the message in *msg_buf* and produces the MIC from it, storing it in *xmit_buf*. The server then sends the MIC back to the client with `send_token()`, and the client verifies it with `gss_verify_mic()`. See "Verifying the Message" on page 72.

Finally, `sign_server()` performs some cleanup; it releases the GSS-API buffers *msg_buf* and *xmit_buf* with `gss_release_buffer()` and then destroys the context with `gss_delete_sec_context()`.

Importing and Exporting a Context

As noted in "Context Export and Import" on page 51, the GSS-API allows you to export and import contexts. The usual reason for doing this is to share a context between different processes in a multiprocess program.

`sign_server()` contains a proof-of-concept function, `test_import_export_context()`, which illustrates how exporting and importing contexts works. This function doesn't pass a context between processes. It only displays the amount of time it takes to export and then import a context. Although rather an artificial function, it does indicate how to use the GSS-API importing and exporting functions, as well as give an idea of how to use timestamps with regard to manipulating contexts. `test_import_export_context()` can be found in "`test_import_export_context()`" on page 106.

Cleanup

Back in the `main()` function, the application deletes the service credential with `gss_delete_cred()` and, if an OID for the mechanism has been specified, deletes that with `gss_delete_oid()` and exits.

Accessory Functions

The client and server programs use certain support functions, for example to display the value of returned flags. As they are either not specific to the GSS-API or else are not terribly important, they are not covered here. They may be found in “Ancillary Functions” on page 108. Two of them, however, `send_token()` and `recv_token()`, are significant enough that they are listed separately in “`send_token()` and `recv_token()`” on page 112.

APPENDIX **A**

Sample C–Based GSS-API Programs

This appendix shows the source code for two sample applications that use GSS-API to make a safe network connection. One application is a client, and the other is a server. The two programs display benchmarks as they run, so that a user can "see" GSS-API being used. Additionally, certain miscellaneous functions are provided for use by the client and server applications. For convenience's sake we have divided each application into its constituent functions.

These programs are examined in detail in Chapter 2.

Client-Side Application

The following sections detail the client-side program, `gss_client`.

Program Headers

These are the declarations for the client program, plus a function that explains the syntax if an incorrect command line is given.

EXAMPLE A–1 Client Program Headers

```
/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
```

EXAMPLE A–1 Client Program Headers *(Continued)*

```
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
 * USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
 * OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
 * PERFORMANCE OF THIS SOFTWARE.
 */


#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <error.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <gssapi/gssapi.h>
#include <gssapi/gssapi_ext.h>
#include "gss-misc.h"

/* global mech oid needed by display status, and acquire cred */
gss_OID g_mechOid = GSS_C_NULL_OID;


void usage()
{
     fprintf(stderr, "Usage: gss-client [-port port] [-d]"
                        " [-mech mechOid] host service msg\n");
     exit(1);
}
```

main()

This is the entry point to the program. The program takes the following syntax on the command line:

```
gss-client [-port port] [-d] [-mech mech] host service msg
```

After parsing the command line, `main()` converts the name of the appropriate security mechanism (if provided) to an OID, establishes a secure connection, and then destroys the mechanism OID, if necessary.

Note – `main()` uses a nonstandard function, `gss_release_oid()`. This function is not supported by all implementations of the GSS-API and should not be used if possible. Since applications should use a default mechanism (specified by `GSS_C_NULL_OID`) instead of allocating one of their own, this function should not be needed in any case. It is included here for reasons of backward compatibility and to show the full extent of this implementation of the GSS-API.

EXAMPLE A–2 `main()`

```
int main(argc, argv)
     int argc;
     char **argv;
{
    /* char *service_name, *hostname, *msg; */
     char *msg;
     char service_name[128];
     char hostname[128];
     char *mechanism = 0;
     u_short port = 4444;
     int use_file = 0;
     OM_uint32 deleg_flag = 0, min_stat;

     display_file = stdout;

     /* Parse arguments. */

        argc--; argv++;
     while (argc) {
          if (strcmp(*argv, "-port") == 0) {
               argc--; argv++;
               if (!argc) usage();
               port = atoi(*argv);
           } else if (strcmp(*argv, "-mech") == 0) {
               argc--; argv++;
               if (!argc) usage();
               mechanism = *argv;
           } else if (strcmp(*argv, "-d") == 0) {
               deleg_flag = GSS_C_DELEG_FLAG;
          } else if (strcmp(*argv, "-f") == 0) {
               use_file = 1;
          } else
                break;
          argc--; argv++;
     }
     if (argc != 3)
          usage();
```

EXAMPLE A–2 `main()` *(Continued)*

```
    if (argc > 1) {
              strcpy(hostname, argv[0]);
       } else if (gethostname(hostname, sizeof(hostname)) == -1) {
                     perror("gethostname");
                     exit(1);
       }


    if (argc > 2) {
       strcpy(service_name, argv[1]);
       strcat(service_name, "@");
       strcat(service_name, hostname);

    }

     msg = argv[2];

    if (mechanism)
        parse_oid(mechanism, &g_mechOid);

    if (call_server(hostname, port, g_mechOid, service_name,
                deleg_flag, msg, use_file) < 0)
         exit(1);

    if (g_mechOid != GSS_C_NULL_OID)
        (void) gss_release_oid(&min_stat, &gmechOid);

    return 0;
}
```

parse_oid()

Converts the name of the security mechanism provided on the command line (if any is provided) to an OID for GSS-API to work with.

Caution – Despite this sample, applications are strongly recommended to use the default mechanism provided by the GSS-API implementation, rather than specifying one. The default mechanism can be obtained by setting the mechanism OID value to `GSS_C_NULL_OID`. Also, the function `gss_str_to_oid()` is not supported by all GSS-API implementations.

EXAMPLE A–3 `parse_oid()`

```
static void parse_oid(char *mechanism, gss_OID *oid)
{
    char        *mechstr = 0, *cp;
```

EXAMPLE A–3 `parse_oid()` *(Continued)*

```
    gss_buffer_desc tok;
    OM_uint32 maj_stat, min_stat;

    if (isdigit(mechanism[0])) {
        mechstr = malloc(strlen(mechanism)+5);
        if (!mechstr) {
            printf("Couldn't allocate mechanism scratch!\n");
            return;
        }
        sprintf(mechstr, "{ %s }", mechanism);
        for (cp = mechstr; *cp; cp++)
            if (*cp == '.')
                *cp = ' ';
        tok.value = mechstr;
    } else
        tok.value = mechanism;
    tok.length = strlen(tok.value);
    maj_stat = gss_str_to_oid(&min_stat, &tok, oid);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("str_to_oid", maj_stat, min_stat);
        return;
    }
    if (mechstr)
        free(mechstr);
}
```

call_server()

This is the centerpiece of the program.

EXAMPLE A–4 `call_server()`

```
/*
 * Function: call_server
 *
 * Purpose: Call the "sign" service.
 *
 * Arguments:
 *
 *      host            (r) the host providing the service
 *      port            (r) the port to connect to on host
 *      service_name    (r) the GSS-API service name to authenticate to
 *      msg             (r) the message to have "signed"
 *
 * Returns: 0 on success, -1 on failure
 *
 * Effects:
 *
 * call_server opens a TCP connection to <host:port> and establishes a
 * GSS-API context with service_name over the connection.  It then
```

EXAMPLE A–4 `call_server()` *(Continued)*

```
 * wraps msg in a GSS-API token with gss_wrap, sends it to the server,
 * reads back a GSS-API signature block for msg from the server, and
 * verifies it with gss_verify.  -1 is returned if any step fails,
 * otherwise 0 is returned.
 */
int call_server(host, port, oid, service_name, deleg_flag, msg, use_file)
     char *host;
     u_short port;
     gss_OID oid;
     char *service_name;
     OM_uint32 deleg_flag;
     char *msg;
     int use_file;
{
     gss_ctx_id_t context;
     gss_buffer_desc in_buf, out_buf, context_token;
     int s, state;
     OM_uint32 ret_flags;
     OM_uint32 maj_stat, min_stat;
     gss_name_t          src_name, targ_name;
     gss_buffer_desc     sname, tname;
     OM_uint32           lifetime;
     gss_OID             mechanism, name_type;
     int                 is_local;
     OM_uint32           context_flags;
     int                 is_open;
     gss_qop_t           qop_state;
     gss_OID_set         mech_names;
     gss_buffer_desc     oid_name;
     int         i;
     int conf_req_flag = 0;
     int req_output_size = 1012;
     OM_uint32 max_input_size = 0;
        char *mechStr;

/* Open connection */
     if ((s = connect_to_server(host, port)) < 0)
          return -1;

     /* Establish context */
     if (client_establish_context(s, service_name, deleg_flag, oid, &context,
                                  &ret_flags) < 0) {
          (void) close(s);
          return -1;
     }

 /* Save and then restore the context */
     maj_stat = gss_export_sec_context(&min_stat,
                                       &context,
                                       &context_token);
     if (maj_stat != GSS_S_COMPLETE) {
             display_status("exporting context", maj_stat, min_stat);
             return -1;
```

EXAMPLE A–4 `call_server()` *(Continued)*

```
    }
    maj_stat = gss_import_sec_context(&min_stat,
                                      &context_token,
                                      &context);
    if (maj_stat != GSS_S_COMPLETE) {
       display_status("importing context", maj_stat, min_stat);
       return -1;
    }
    (void) gss_release_buffer(&min_stat, &context_token);

    /* display the flags */
    display_ctx_flags(ret_flags);

    /* Get context information */
    maj_stat = gss_inquire_context(&min_stat, context,
                                   &src_name, &targ_name, &lifetime,
                                   &mechanism, &context_flags,
                                   &is_local,
                                   &is_open);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("inquiring context", maj_stat, min_stat);
        return -1;
    }

    if (maj_stat == GSS_S_CONTEXT_EXPIRED) {
    printf(" context expired\n");
        display_status("Context is expired", maj_stat, min_stat);
        return -1;
    }

/* Test gss_wrap_size_limit */
    maj_stat = gss_wrap_size_limit(&min_stat, context,
                                   conf_req_flag,
                                   GSS_C_QOP_DEFAULT,
                                   req_output_size,
                                   &max_input_size
                                   );
    if (maj_stat != GSS_S_COMPLETE) {
          display_status("wrap_size_limit call", maj_stat, min_stat);
    } else
          fprintf (stderr, "gss_wrap_size_limit returned "
               "max input size = %d \n"
               "for req_output_size = %d with Integrity only\n",
               max_input_size , req_output_size , conf_req_flag);

    conf_req_flag = 1;
    maj_stat = gss_wrap_size_limit(&min_stat, context,
                                   conf_req_flag,
                                   GSS_C_QOP_DEFAULT,
                                   req_output_size,
                                   &max_input_size
                                   );
    if (maj_stat != GSS_S_COMPLETE) {
```

EXAMPLE A–4 `call_server()` *(Continued)*

```
      display_status("wrap_size_limit call", maj_stat, min_stat);
} else
      fprintf (stderr, "gss_wrap_size_limit returned "
           " max input size = %d \n"
           "for req_output_size = %d with "
           "Integrity & Privacy \n",
           max_input_size , req_output_size );

maj_stat = gss_display_name(&min_stat, src_name, &sname,
                            &name_type);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("displaying source name", maj_stat, min_stat);
    return -1;
}
maj_stat = gss_display_name(&min_stat, targ_name, &tname,
                            (gss_OID *) NULL);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("displaying target name", maj_stat, min_stat);
    return -1;
}
fprintf(stderr, "\"%.*s\" to \"%.*s\", lifetime %u, flags %x, %s, %s\n",
        (int) sname.length, (char *) sname.value,
        (int) tname.length, (char *) tname.value, lifetime,
        context_flags,
        (is_local) ? "locally initiated" : "remotely initiated",
        (is_open) ? "open" : "closed");

(void) gss_release_name(&min_stat, &src_name);
(void) gss_release_name(&min_stat, &targ_name);
(void) gss_release_buffer(&min_stat, &sname);
(void) gss_release_buffer(&min_stat, &tname);


maj_stat = gss_oid_to_str(&min_stat,
                          name_type,
                          &oid_name);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("converting oid->string", maj_stat, min_stat);
    return -1;
}
fprintf(stderr, "Name type of source name is %.*s.\n",
        (int) oid_name.length, (char *) oid_name.value);
(void) gss_release_buffer(&min_stat, &oid_name);

/* Now get the names supported by the mechanism */
maj_stat = gss_inquire_names_for_mech(&min_stat,
                                      mechanism,
                                      &mech_names);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("inquiring mech names", maj_stat, min_stat);
    return -1;
}
```

EXAMPLE A–4 `call_server()` *(Continued)*

```
maj_stat = gss_oid_to_str(&min_stat,
                          mechanism,
                          &oid_name);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("converting oid->string", maj_stat, min_stat);
    return -1;
}
   mechStr = (char *)__gss_oid_to_mech(mechanism);
fprintf(stderr, "Mechanism %.*s (%s) supports %d names\n",
           (int) oid_name.length, (char *) oid_name.value,
           (mechStr == NULL ? "NULL" : mechStr),
           mech_names->count);
(void) gss_release_buffer(&min_stat, &oid_name);

for (i=0; i < mech_names->count; i++) {
    maj_stat = gss_oid_to_str(&min_stat,
                              &mech_names->elements[i],
                              &oid_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
    }
    fprintf(stderr, "  %d: %.*s\n", i,
            (int) oid_name.length, (char *) oid_name.value);

    (void) gss_release_buffer(&min_stat, &oid_name);
}
(void) gss_release_oid_set(&min_stat, &mech_names);

if (use_file) {
    read_file(msg, &in_buf);
} else {
    /* Seal the message */
    in_buf.value = msg;
    in_buf.length = strlen(msg) + 1;
}

if (ret_flag & GSS_C_CONF_FLAG) {
     state = 1;
else
     state = 0;
}

maj_stat = gss_wrap(&min_stat, context, 1, GSS_C_QOP_DEFAULT,
                    &in_buf, &state, &out_buf);
if (maj_stat != GSS_S_COMPLETE) {
     display_status("wrapping message", maj_stat, min_stat);
     (void) close(s);
     (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
     return -1;
} else if (! state) {
     fprintf(stderr, "Warning!  Message not encrypted.\n");
}
```

EXAMPLE A–4 `call_server()` *(Continued)*

```
    /* Send to server */
    if (send_token(s, &out_buf) < 0) {
         (void) close(s);
         (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
         return -1;
    }
    (void) gss_release_buffer(&min_stat, &out_buf);

    /* Read signature block into out_buf */
    if (recv_token(s, &out_buf) < 0) {
         (void) close(s);
         (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
         return -1;
    }

    /* Verify signature block */
    maj_stat = gss_verify_mic(&min_stat, context, &in_buf,
                              &out_buf, &qop_state);
    if (maj_stat != GSS_S_COMPLETE) {
         display_status("verifying signature", maj_stat, min_stat);
         (void) close(s);
         (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
         return -1;
    }
    (void) gss_release_buffer(&min_stat, &out_buf);

    if (use_file)
        free(in_buf.value);

    printf("Signature verified.\n");

    /* Delete context */
    maj_stat = gss_delete_sec_context(&min_stat, &context, &out_buf);
    if (maj_stat != GSS_S_COMPLETE) {
         display_status("deleting context", maj_stat, min_stat);
         (void) close(s);
         (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
         return -1;
    }

    (void) gss_release_buffer(&min_stat, &out_buf);
    (void) close(s);
    return 0;
}
```

`read_file()`

In the case that the message to be transferred is contained in a file, this function, called by `call_server()`, opens and reads the file.

EXAMPLE A–5 read_file()

```
void read_file(file_name, in_buf)
    char                *file_name;
    gss_buffer_t        in_buf;
{
    int fd, bytes_in, count;
    struct stat stat_buf;

    if ((fd = open(file_name, O_RDONLY, 0)) < 0) {
        perror("open");
        fprintf(stderr, "Couldn't open file %s\n", file_name);
        exit(1);
    }
    if (fstat(fd, &stat_buf) < 0) {
        perror("fstat");
        exit(1);
    }
    in_buf->length = stat_buf.st_size;
    in_buf->value = malloc(in_buf->length);
    if (in_buf->value == 0) {
        fprintf(stderr, "Couldn't allocate %ld byte buffer for reading file\n",
                in_buf->length);
        exit(1);
    }
    memset(in_buf->value, 0, in_buf->length);
    for (bytes_in = 0; bytes_in < in_buf->length; bytes_in += count) {
        count = read(fd, in_buf->value, (OM_uint32)in_buf->length);
        if (count < 0) {
            perror("read");
            exit(1);
        }
        if (count == 0)
            break;
    }
    if (bytes_in != count)
        fprintf(stderr, "Warning, only read in %d bytes, expected %d\n",
                bytes_in, count);
}
```

client_establish_context()

Calls gss_init_sec_context() to establish a context with the server.

EXAMPLE A–6 client_establish_context()

```
/*
 * Function: client_establish_context
 *
 * Purpose: establishes a GSS-API context with a specified service and
 * returns the context handle
 *
 * Arguments:
```

EXAMPLE A–6 client_establish_context() *(Continued)*

```
 *
 *      s               (r) an established TCP connection to the service
 *      service_name    (r) the ASCII service name of the service
 *      context         (w) the established GSS-API context
 *      ret_flags       (w) the returned flags from init_sec_context
 *
 * Returns: 0 on success, -1 on failure
 *
 * Effects:
 *
 * service_name is imported as a GSS-API name and a GSS-API context is
 * established with the corresponding service; the service should be
 * listening on the TCP connection s.  The default GSS-API mechanism
 * is used, and mutual authentication and replay detection are
 * requested.
 *
 * If successful, the context handle is returned in context.  If
 * unsuccessful, the GSS-API error messages are displayed on stderr
 * and -1 is returned.
 */
    int client_establish_context(s, service_name, deleg_flag, oid,
                                 gss_context, ret_flags)
    int s;
    char *service_name;
    gss_OID oid;
    OM_uint32 deleg_flag;
    gss_ctx_id_t *gss_context;
    OM_uint32 *ret_flags;
{
    gss_buffer_desc send_tok, recv_tok, *token_ptr;
    gss_name_t target_name;
    OM_uint32 maj_stat, min_stat;

    /*
     * Import the name into target_name.  Use send_tok to save
     * local variable space.
     */

    send_tok.value = service_name;
    send_tok.length = strlen(service_name) + 1;
    maj_stat = gss_import_name(&min_stat, &send_tok,
                       (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &target_name);
    if (maj_stat != GSS_S_COMPLETE) {
         display_status("parsing name", maj_stat, min_stat);
         return -1;
    }


    /*
     * Perform the context-establishement loop.
     *
     * On each pass through the loop, token_ptr points to the token
     * to send to the server (or GSS_C_NO_BUFFER on the first pass).
```

EXAMPLE A–6 client_establish_context() *(Continued)*

```
 * Every generated token is stored in send_tok which is then
 * transmitted to the server; every received token is stored in
 * recv_tok, which token_ptr is then set to, to be processed by
 * the next call to gss_init_sec_context.
 *
 * GSS-API guarantees that send_tok's length will be non-zero
 * if and only if the server is expecting another token from us,
 * and that gss_init_sec_context returns GSS_S_CONTINUE_NEEDED if
 * and only if the server has another token to send us.
 */

token_ptr = GSS_C_NO_BUFFER;
*gss_context = GSS_C_NO_CONTEXT;

do {
     maj_stat =
          gss_init_sec_context(&min_stat,
                               GSS_C_NO_CREDENTIAL,
                               gss_context,
                               target_name,
                               oid,
                               GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG |
                                                   deleg_flag,
                               0,
                               NULL,       /* no channel bindings */
                               token_ptr,
                               NULL,       /* ignore mech type */
                               &send_tok,
                               ret_flags,
                               NULL);      /* ignore time_rec */
     if (gss_context == NULL){
          printf("Cannot create context\n");
          return GSS_S_NO_CONTEXT;
     }
     if (token_ptr != GSS_C_NO_BUFFER)
          (void) gss_release_buffer(&min_stat, &recv_tok);
     if (maj_stat!=GSS_S_COMPLETE && maj_stat!=GSS_S_CONTINUE_NEEDED) {
          display_status("initializing context", maj_stat, min_stat);
          (void) gss_release_name(&min_stat, &target_name);
          return -1;
     }

     if (send_tok.length != 0) {
          fprintf(stdout, "Sending init_sec_context token (size=%ld)...",
                    send_tok.length);
          if (send_token(s, &send_tok) < 0) {
               (void) gss_release_buffer(&min_stat, &send_tok);
               (void) gss_release_name(&min_stat, &target_name);
               return -1;
          }
     }
     (void) gss_release_buffer(&min_stat, &send_tok);
```

EXAMPLE A–6 client_establish_context() *(Continued)*

```
          if (maj_stat == GSS_S_CONTINUE_NEEDED) {
               fprintf(stdout, "continue needed...");
               if (recv_token(s, &recv_tok) < 0) {
                    (void) gss_release_name(&min_stat, &target_name);
                    return -1;
               }
               token_ptr = &recv_tok;
          }
          printf("\n");
     } while (maj_stat == GSS_S_CONTINUE_NEEDED);

     (void) gss_release_name(&min_stat, &target_name);
     return 0;
}
```

connect_to_server()

This offers a basic, no-frills function that creates a TCP connection.

EXAMPLE A–7 connect_to_server()

```
/*
 * Function: connect_to_server
 *
 * Purpose: Opens a TCP connection to the name host and port.
 *
 * Arguments:
 *
 *      host            (r) the target host name
 *      port            (r) the target port, in host byte order
 *
 * Returns: the established socket file desciptor, or -1 on failure
 *
 * Effects:
 *
 * The host name is resolved with gethostbyname(), and the socket is
 * opened and connected.  If an error occurs, an error message is
 * displayed and -1 is returned.
 */
int connect_to_server(host, port)
     char *host;
     u_short port;
{
     struct sockaddr_in saddr;
     struct hostent *hp;
     int s;

     if ((hp = gethostbyname(host)) == NULL) {
          fprintf(stderr, "Unknown host: %s\n", host);
          return -1;
```

EXAMPLE A–7 connect_to_server() *(Continued)*

```
    }

    saddr.sin_family = hp->h_addrtype;
    memcpy((char *)&saddr.sin_addr, hp->h_addr, sizeof(saddr.sin_addr));
    saddr.sin_port = htons(port);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
         perror("creating socket");
         return -1;
    }
    if (connect(s, (struct sockaddr *)&saddr, sizeof(saddr)) < 0) {
         perror("connecting to server");
         (void) close(s);
         return -1;
    }

    return s;
}
```

Server-Side Application

This is the application that receives messages from the client function described earlier.

Program Headers

These are the declarations for the server program, plus a function that explains the syntax if an incorrect command line is given. Here the security mechanism is set to be the GSS-API-provided default.

EXAMPLE A–8 Program Headers

```
/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
```

EXAMPLE A–8 Program Headers *(Continued)*

```
 *
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
 * USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
 * OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
 * PERFORMANCE OF THIS SOFTWARE.
 */

#if !defined(lint) && !defined(__CODECENTER__)
static char *rcsid = "$Header: /afs/athena.mit.edu/astaff/project/krbdev/.cvsroot
/src/appl/gss-sample/gss-server.c,v 1.17 1996/10/22 00:07:59 tytso Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#include <stdlib.h>
#include <ctype.h>

#include <gssapi/gssapi.h>
#include <gssapi/gssapi_ext.h>
#include "gss-misc.h"

#ifdef USE_STRING_H
#include <string.h>
#else
#include <strings.h>
#endif

/* global mechanism oid used in acquire cred and display status */
gss_OID g_mechOid = GSS_C_NULL_OID;

void usage()
{
     fprintf(stderr, "Usage: gss-server [-port port] [-verbose]\n");
     fprintf(stderr, "       [-inetd] [-logfile file]");
     fprintf(stderr, " [-mech mechoid] [service_name]\n");
     exit(1);
}

FILE *log;

int verbose = 0;
```

main()

This is the entry point to the program. The program takes the following syntax on the command line:

`gss-server [-port` *port*`] [-d] [-mech` *mech*`]` *host service msg*

After parsing the command line, `main()` converts the name of the desired security mechanism (if provided) to an OID, acquires credentials, establishes a context and receives data, and then destroys the mechanism OID if necessary.

Note – Applications should normally not set the mechanism, but use defaults provided by the GSS-API.

EXAMPLE A–9 `main()`

```
int
main(argc, argv)
     int argc;
     char **argv;
{
     char *service_name, *mechType = NULL;
     gss_cred_id_t server_creds;
     OM_uint32 min_stat;
     u_short port = 4444;
     int s;
     int once = 0;
     int do_inetd = 0;

     log = stdout;
     display_file = stdout;
     argc--; argv++;
     while (argc) {
          if (strcmp(*argv, "-port") == 0) {
               argc--; argv++;
               if (!argc) usage();
               port = atoi(*argv);
          } else if (strcmp(*argv, "-verbose") == 0) {
              verbose = 1;
          } else if (strcmp(*argv, "-once") == 0) {
              once = 1;
          } else if (strcmp(*argv, "-inetd") == 0) {
              do_inetd = 1;
          } else if (strcmp(*argv, "-mech") == 0) {
                argc--; argv++;
                if (!argc)      usage();
                mechType = *argv;
          } else if (strcmp(*argv, "-logfile") == 0) {
              argc--; argv++;
              if (!argc) usage();
              log = fopen(*argv, "a");
```

EXAMPLE A–9 `main()` *(Continued)*

```
            display_file = log;
            if (!log) {
                perror(*argv);
                exit(1);
            }
        } else
             break;
        argc--; argv++;
    }
    if (argc != 1)
        usage();

    if ((*argv)[0] == '-')
        usage();

    service_name = *argv;

    if (mechType != NULL) {
            if ((g_mechOid = createMechOid(mechType)) == NULL) {
                    usage();
                    exit(-1);
            }
    }

    if (server_acquire_creds(service_name, g_mechOid, &server_creds) < 0)
        return -1;

    if (do_inetd) {
        close(1);
        close(2);

        sign_server(0, server_creds);
        close(0);
    } else {
        int stmp;

        if ((stmp = create_socket(port))) {
            do {
                /* Accept a TCP connection */
                if ((s = accept(stmp, NULL, 0)) < 0) {
                    perror("accepting connection");
                } else {
                    /* this return value is not checked, because there's
                       not really anything to do if it fails */
                    sign_server(s, server_creds);
                }
            } while (!once);
        }

        close(stmp);
    }

    (void) gss_release_cred(&min_stat, &server_creds);
```

EXAMPLE A–9 `main()` *(Continued)*

```
     if (g_mechOid != GSS_C_NULL_OID)
             gss_release_oid(&min_stat, &g_mechOid);

     /*NOTREACHED*/
     (void) close(s);
     return 0;
}
```

createMechOid()

This function is shown for completeness' sake. Normally, you should use the default mechanism (specified by GSS_C_NULL_OID).

EXAMPLE A–10 `createMechOid()`

```
gss_OID createMechOid(const char *mechStr)
{
        gss_buffer_desc mechDesc;
        gss_OID mechOid;
        OM_uint32 minor;

        if (mechStr == NULL)
                return (GSS_C_NULL_OID);

        mechDesc.length = strlen(mechStr);
        mechDesc.value = (void *) mechStr;

        if (gss_str_to_oid(&minor, &mechDesc, &mechOid) !
= GSS_S_COMPLETE) {
                fprintf(stderr, "Invalid mechanism oid specified <%s>",
                                mechStr);
                return (GSS_C_NULL_OID);
        }

        return (mechOid);
}
```

server_acquire_creds()

Gets the credentials for the requested network service.

EXAMPLE A–11 `server_acquire_creds()`

```
/*
 * Function: server_acquire_creds
 *
 * Purpose: imports a service name and acquires credentials for it
```

EXAMPLE A–11 server_acquire_creds() *(Continued)*

```
 *
 * Arguments:
 *
 *      service_name    (r) the ASCII service name
        mechType        (r) the mechanism type to use
 *      server_creds    (w) the GSS-API service credentials
 *
 * Returns: 0 on success, -1 on failure
 *
 * Effects:
 *
 * The service name is imported with gss_import_name, and service
 * credentials are acquired with gss_acquire_cred.  If either operation
 * fails, an error message is displayed and -1 is returned; otherwise,
 * 0 is returned.
 */
int server_acquire_creds(service_name, mechOid, server_creds)
     char *service_name;
     gss_OID mechOid;
     gss_cred_id_t *server_creds;
{
     gss_buffer_desc name_buf;
     gss_name_t server_name;
     OM_uint32 maj_stat, min_stat;
     gss_OID_set_desc mechOidSet;
     gss_OID_set desiredMechs = GSS_C_NULL_OID_SET;

     if (mechOid != GSS_C_NULL_OID) {
               desiredMechs = &mechOidSet;
               mechOidSet.count = 1;
               mechOidSet.elements = mechOid;
     } else
               desiredMechs = GSS_C_NULL_OID_SET;


     name_buf.value = service_name;
     name_buf.length = strlen(name_buf.value) + 1;
     maj_stat = gss_import_name(&min_stat, &name_buf,
               (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &server_name);
     if (maj_stat != GSS_S_COMPLETE) {
          display_status("importing name", maj_stat, min_stat);
          if (mechOid != GSS_C_NO_OID)
               gss_release_oid(&min_stat, &mechOid);
          return -1;
     }

     maj_stat = gss_acquire_cred(&min_stat, server_name, 0,
                                 desiredMechs, GSS_C_ACCEPT,
                                 server_creds, NULL, NULL);

     if (maj_stat != GSS_S_COMPLETE) {
          display_status("acquiring credentials", maj_stat, min_stat);
          return -1;
```

EXAMPLE A–11 `server_acquire_creds()` *(Continued)*

```
     }

     (void) gss_release_name(&min_stat, &server_name);

     return 0;
}
```

sign_server()

This is the "guts" of the program. Calls `server_establish_context()` to accept the context, receives the data, unwraps it, verifies it, then generates a MIC to send back to the client. Finally, it deletes the context.

EXAMPLE A–12 `sign_server()`

```
/*
 * Function: sign_server
 *
 * Purpose: Performs the "sign" service.
 *
 * Arguments:
 *
 *      s               (r) a TCP socket on which a connection has been
 *                      accept()ed
 *      service_name    (r) the ASCII name of the GSS-API service to
 *                      establish a context as
 *
 * Returns: -1 on error
 *
 * Effects:
 *
 * sign_server establishes a context, and performs a single sign request.
 *
 * A sign request is a single GSS-API wrapped token.  The token is
 * unwrapped and a signature block, produced with gss_get_mic, is returned
 * to the sender.  The context is the destroyed and the connection
 * closed.
 *
 * If any error occurs, -1 is returned.
 */
int sign_server(s, server_creds)
     int s;
     gss_cred_id_t server_creds;
{
     gss_buffer_desc client_name, xmit_buf, msg_buf;
     gss_ctx_id_t context;
     OM_uint32 maj_stat, min_stat;
     int i, conf_state, ret_flags;
     char       *cp;
```

EXAMPLE A–12 `sign_server()` *(Continued)*

```
/* Establish a context with the client */
if (server_establish_context(s, server_creds, &context,
                             &client_name, &ret_flags) < 0)
   return(-1);

printf("Accepted connection: \"%.*s\"\n",
       (int) client_name.length, (char *) client_name.value);
(void) gss_release_buffer(&min_stat, &client_name);

for (i=0; i < 3; i++)
        if (test_import_export_context(&context))
                return -1;

/* Receive the wrapped message token */
if (recv_token(s, &xmit_buf) < 0)
   return(-1);

if (verbose && log) {
   fprintf(log, "Wrapped message token:\n");
   print_token(&xmit_buf);
}

maj_stat = gss_unwrap(&min_stat, context, &xmit_buf, &msg_buf,
                      &conf_state, (gss_qop_t *) NULL);
if (maj_stat != GSS_S_COMPLETE) {
   display_status("unwrapping message", maj_stat, min_stat);
   return(-1);
} else if (! conf_state) {
   fprintf(stderr, "Warning!  Message not encrypted.\n");
}

(void) gss_release_buffer(&min_stat, &xmit_buf);

fprintf(log, "Received message: ");
cp = msg_buf.value;
if (isprint(cp[0]) && isprint(cp[1]))
   fprintf(log, "\"%s\"\n", cp);
else {
   printf("\n");
   print_token(&msg_buf);
}

/* Produce a signature block for the message */
maj_stat = gss_get_mic(&min_stat, context, GSS_C_QOP_DEFAULT,
                       &msg_buf, &xmit_buf);
if (maj_stat != GSS_S_COMPLETE) {
   display_status("signing message", maj_stat, min_stat);
   return(-1);
}

(void) gss_release_buffer(&min_stat, &msg_buf);

/* Send the signature block to the client */
```

EXAMPLE A–12 sign_server() *(Continued)*

```
     if (send_token(s, &xmit_buf) < 0)
        return(-1);

     (void) gss_release_buffer(&min_stat, &xmit_buf);

     /* Delete context */
     maj_stat = gss_delete_sec_context(&min_stat, &context, NULL);
     if (maj_stat != GSS_S_COMPLETE) {
        display_status("deleting context", maj_stat, min_stat);
        return(-1);
     }

     fflush(log);

     return(0);
}
```

server_establish_context()

This calls gss_accept_sec_context() as part of a context-establishment loop.

EXAMPLE A–13 server_establish_context()

```
/*
 * Function: server_establish_context
 *
 * Purpose: establishses a GSS-API context as a specified service with
 * an incoming client, and returns the context handle and associated
 * client name
 *
 * Arguments:
 *
 *      s               (r) an established TCP connection to the client
 *      service_creds   (r) server credentials, from gss_acquire_cred
 *      context         (w) the established GSS-API context
 *      client_name     (w) the client's ASCII name
 *
 * Returns: 0 on success, -1 on failure
 *
 * Effects:
 *
 * Any valid client request is accepted.  If a context is established,
 * its handle is returned in context and the client name is returned
 * in client_name and 0 is returned.  If unsuccessful, an error
 * message is displayed and -1 is returned.
 */
int server_establish_context(s, server_creds, context, client_name, ret_flags)
     int s;
     gss_cred_id_t server_creds;
     gss_ctx_id_t *context;
```

EXAMPLE A–13 server_establish_context() *(Continued)*

```
     gss_buffer_t client_name;
     OM_uint32 *ret_flags;
{
     gss_buffer_desc send_tok, recv_tok;
     gss_name_t client;
     gss_OID doid;
     OM_uint32 maj_stat, min_stat;
     gss_buffer_desc    oid_name;
        char *mechStr;

     *context = GSS_C_NO_CONTEXT;

     do {
          if (recv_token(s, &recv_tok) < 0)
               return -1;

          if (verbose && log) {
              fprintf(log, "Received token (size=%d): \n", recv_tok.length);
              print_token(&recv_tok);
          }

          maj_stat =
               gss_accept_sec_context(&min_stat,
                                      context,
                                      server_creds,
                                      &recv_tok,
                                      GSS_C_NO_CHANNEL_BINDINGS,
                                      &client,
                                      &doid,
                                      &send_tok,
                                      ret_flags,
                                      NULL,     /* ignore time_rec */
                                      NULL);    /* ignore del_cred_handle */

          if (maj_stat!=GSS_S_COMPLETE && maj_stat!=GSS_S_CONTINUE_NEEDED) {
               display_status("accepting context", maj_stat, min_stat);
               (void) gss_release_buffer(&min_stat, &recv_tok);
               return -1;
          }

          (void) gss_release_buffer(&min_stat, &recv_tok);

          if (send_tok.length != 0) {
              if (verbose && log) {
                  fprintf(log,
                          "Sending accept_sec_context token (size=%d):\n",
                          send_tok.length);
                  print_token(&send_tok);
              }
               if (send_token(s, &send_tok) < 0) {
                    fprintf(log, "failure sending token\n");
                    return -1;
               }
```

EXAMPLE A–13 server_establish_context() *(Continued)*

```
            (void) gss_release_buffer(&min_stat, &send_tok);
        }
        if (verbose && log) {
            if (maj_stat == GSS_S_CONTINUE_NEEDED)
                fprintf(log, "continue needed...\n");
            else
                fprintf(log, "\n");
            fflush(log);
        }
    } while (maj_stat == GSS_S_CONTINUE_NEEDED);

    /* display the flags */
    display_ctx_flags(*ret_flags);

    if (verbose && log) {
        maj_stat = gss_oid_to_str(&min_stat, doid, &oid_name);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("converting oid->string", maj_stat, min_stat);
            return -1;
        }
              mechStr = (char *)__gss_oid_to_mech(doid);
        fprintf(log, "Accepted connection using mechanism OID %.*s (%s).\n",
                (int) oid_name.length, (char *) oid_name.value,
                       (mechStr == NULL ? "NULL" : mechStr));
        (void) gss_release_buffer(&min_stat, &oid_name);
    }

    maj_stat = gss_display_name(&min_stat, client, client_name, &doid);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("displaying name", maj_stat, min_stat);
        return -1;
    }
    return 0;
}
```

create_a_socket()

This is a no-frills function for creating a transport connection with the client.

EXAMPLE A–14 create_a_socket()

```
/*
 * Function: create_socket
 *
 * Purpose: Opens a listening TCP socket.
 *
 * Arguments:
 *
 *      port            (r) the port number on which to listen
```

EXAMPLE A–14 `create_a_socket()` *(Continued)*

```
 *
 * Returns: the listening socket file descriptor, or -1 on failure
 *
 * Effects:
 *
 * A listening socket on the specified port and created and returned.
 * On error, an error message is displayed and -1 is returned.
 */
int create_socket(port)
     u_short port;
{
     struct sockaddr_in saddr;
     int s;
     int on = 1;

     saddr.sin_family = AF_INET;
     saddr.sin_port = htons(port);
     saddr.sin_addr.s_addr = INADDR_ANY;

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("creating socket");
          return -1;
     }
     /* Let the socket be reused right away */
     (void) setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&on,
                                    sizeof(on));
     if (bind(s, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
{
          perror("binding socket");
          (void) close(s);
          return -1;
     }
     if (listen(s, 5) < 0) {
          perror("listening on socket");
          (void) close(s);
          return -1;
     }
     return s;
}
```

test_import_export_context()

Finally, this is a small function to show how `gss_export_sec_context()` and `gss_import_sec_context()` work. Of limited practicality, this function is here mostly to indicate how these GSS-API functions can be used.

EXAMPLE A–15 `test_import_export_context()`

```
int test_import_export_context(context)
        gss_ctx_id_t *context;
{
```

EXAMPLE A–15 `test_import_export_context()` *(Continued)*

```
        OM_uint32       min_stat, maj_stat;
        gss_buffer_desc context_token, copied_token;
        struct timeval tm1, tm2;

        /*
         * Attempt to save and then restore the context.
         */
        gettimeofday(&tm1, (struct timezone *)0);
        maj_stat = gss_export_sec_context(&min_stat, context, &context_token);
        if (maj_stat != GSS_S_COMPLETE) {
                display_status("exporting context", maj_stat, min_stat);
                return 1;
        }
        gettimeofday(&tm2, (struct timezone *)0);
        if (verbose && log)
                fprintf(log, "Exported context: %d bytes, %7.4f seconds\n",
                        context_token.length, timeval_subtract(&tm2, &tm1));
        copied_token.length = context_token.length;
        copied_token.value = malloc(context_token.length);
        if (copied_token.value == 0) {
            fprintf(log, "Couldn't allocate memory to copy context token.\n");
            return 1;
        }
        memcpy(copied_token.value, context_token.value, copied_token.length);
        maj_stat = gss_import_sec_context(&min_stat, &copied_token, context);
        if (maj_stat != GSS_S_COMPLETE) {
                display_status("importing context", maj_stat, min_stat);
                return 1;
        }
        gettimeofday(&tm1, (struct timezone *)0);
        if (verbose && log)
                fprintf(log, "Importing context: %7.4f seconds\n",
                        timeval_subtract(&tm1, &tm2));
        (void) gss_release_buffer(&min_stat, &context_token);
        return 0;
}
```

`timeval_subtract()`

This is a convenience function used by `test_import_export_context()`.

EXAMPLE A–16 `timeval_subtract()`

```
static float timeval_subtract(tv1, tv2)
        struct timeval *tv1, *tv2;
{
        return ((tv1->tv_sec - tv2->tv_sec) +
                ((float) (tv1->tv_usec - tv2->tv_usec)) / 1000000);
}
```

Ancillary Functions

To make the client and server programs work as shown, a number of other functions are required. These are mostly for displaying values, and are not necessary to the basic functioning of the programs. They are shown here for completeness.

Two functions, however, are significant: `send_token()` and `recv_token()`, which do the actual transfer of context tokens and messages. They are actually plain "vanilla" functions that open up a file descriptor and read to or write from it. Although ordinary, and not directly related to the GSS-API, they are sufficiently important to call out separately.

Miscellaneous Support Functions

These functions include:

- `display_status()` — Shows the status returned by the last GSS-API function called.
- `write_all()` — Writes a buffer to a file.
- `read_all()` — Reads a file into a buffer.
- `display_ctx_flags()` — Shows in a readable form information about the current context, such as whether confidentiality or mutual authentication is allowed.
- `print_token()` — Prints out a token's value.

EXAMPLE A–17 Miscellaneous GSS-API Support Functions

```
/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
 * USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
 * OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
```

EXAMPLE A–17 Miscellaneous GSS-API Support Functions *(Continued)*

```
 * PERFORMANCE OF THIS SOFTWARE.
 */

#if !defined(lint) && !defined(__CODECENTER__)
static char *rcsid = "$Header: /afs/athena.mit.edu/astaff/project/krbdev/.cvsroot
/src/appl/gss-sample/gss-misc.c,v 1.15 1996/07/22 20:21:20 marc Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <errno.h>
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#include <string.h>

#include <gssapi/gssapi.h>
#include "gss-misc.h"
#include <stdlib.h>

FILE *display_file;
extern gss_OID g_mechOid;


static void display_status_1(char *m, OM_uint32 code, int type);

static int write_all(int fildes, char *buf, unsigned int nbyte)
{
     int ret;
     char *ptr;

     for (ptr = buf; nbyte; ptr += ret, nbyte -= ret) {
          ret = write(fildes, ptr, nbyte);
          if (ret < 0) {
               if (errno == EINTR)
                    continue;
               return(ret);
          } else if (ret == 0) {
               return(ptr-buf);
          }
     }

     return(ptr-buf);
}

static int read_all(int fildes, char *buf, unsigned int nbyte)
{
     int ret;
     char *ptr;

     for (ptr = buf; nbyte; ptr += ret, nbyte -= ret) {
          ret = read(fildes, ptr, nbyte);
```

EXAMPLE A–17 Miscellaneous GSS-API Support Functions *(Continued)*

```
          if (ret < 0) {
               if (errno == EINTR)
                    continue;
               return(ret);
          } else if (ret == 0) {
               return(ptr-buf);
          }
     }

     return(ptr-buf);
}

static void display_status_1(m, code, type)
     char *m;
     OM_uint32 code;
     int type;
{
     OM_uint32 maj_stat, min_stat;
     gss_buffer_desc msg = GSS_C_EMPTY_BUFFER;
     OM_uint32 msg_ctx;

     msg_ctx = 0;
     while (1) {
          maj_stat = gss_display_status(&min_stat, code,
                                        type, g_mechOid,
                                        &msg_ctx, &msg);
          if (maj_stat != GSS_S_COMPLETE) {
                  if (display_file) {
                        fprintf(display_file, "error in gss_display_status"
                                        " called from <%s>\n", m);
                  }
                  break;
          }
          else if (display_file)
                fprintf(display_file, "GSS-API error %s: %s\n", m,
                      (char *)msg.value);
          if (msg.length != 0)
                (void) gss_release_buffer(&min_stat, &msg);

          if (!msg_ctx)
               break;
     }
}

/*
 * Function: display_status
 *
 * Purpose: displays GSS-API messages
 *
 * Arguments:
 *
 *      msg             a string to be displayed with the message
 *      maj_stat        the GSS-API major status code
```

EXAMPLE A–17 Miscellaneous GSS-API Support Functions *(Continued)*

```
 *      min_stat        the GSS-API minor status code
 *
 * Effects:
 *
 * The GSS-API messages associated with maj_stat and min_stat are
 * displayed on stderr, each preceeded by "GSS-API error <msg>:
" and
 * followed by a newline.
 */
void display_status(msg, maj_stat, min_stat)
     char *msg;
     OM_uint32 maj_stat;
     OM_uint32 min_stat;
{
     display_status_1(msg, maj_stat, GSS_C_GSS_CODE);
     display_status_1(msg, min_stat, GSS_C_MECH_CODE);
}


/*
 * Function: display_ctx_flags
 *
 * Purpose: displays the flags returned by context initation in
 *          a human-readable form
 *
 * Arguments:
 *
 *      int             ret_flags
 *
 * Effects:
 *
 * Strings corresponding to the context flags are printed on
 * stdout, preceded by "context flag: " and followed by a newline
 */

void display_ctx_flags(flags)
     OM_uint32 flags;
{
     if (flags & GSS_C_DELEG_FLAG)
          fprintf(display_file, "context flag: GSS_C_DELEG_FLAG\n");
     if (flags & GSS_C_MUTUAL_FLAG)
          fprintf(display_file, "context flag: GSS_C_MUTUAL_FLAG\n");
     if (flags & GSS_C_REPLAY_FLAG)
          fprintf(display_file, "context flag: GSS_C_REPLAY_FLAG\n");
     if (flags & GSS_C_SEQUENCE_FLAG)
          fprintf(display_file, "context flag: GSS_C_SEQUENCE_FLAG\n");
     if (flags & GSS_C_CONF_FLAG )
          fprintf(display_file, "context flag: GSS_C_CONF_FLAG \n");
     if (flags & GSS_C_INTEG_FLAG )
          fprintf(display_file, "context flag: GSS_C_INTEG_FLAG \n");
}

void print_token(tok)
```

EXAMPLE A–17 Miscellaneous GSS-API Support Functions *(Continued)*

```
     gss_buffer_t tok;
{
    int i;
    unsigned char *p = tok->value;

    if (!display_file)
        return;
    for (i=0; i < tok->length; i++, p++) {
        fprintf(display_file, "%02x ", *p);
        if ((i % 16) == 15) {
            fprintf(display_file, "\n");
        }
    }
    fprintf(display_file, "\n");
    fflush(display_file);
}
```

`send_token()` and `recv_token()`

These functions send and receive data between the client and the server. (In a multiprocess application they could do the same between processes.) They are slightly misnamed, since they send and receive *messages* as well as *tokens*. They are oblivious to the content they handle.

`send_token()`

This function sends a token or message.

EXAMPLE A–18 `send_token()`

```
/*
 * Function: send_token
 *
 * Purpose: Writes a token to a file descriptor.
 *
 * Arguments:
 *
 *      s               (r) an open file descriptor
 *      tok             (r) the token to write
 *
 * Returns: 0 on success, -1 on failure
 *
 * Effects:
 *
 * send_token writes the token length (as a network long) and then the
 * token data to the file descriptor s.  It returns 0 on success, and
 * -1 if an error occurs or if it could not write all the data.
```

EXAMPLE A–18 send_token() *(Continued)*

```
 */
int send_token(s, tok)
     int s;
     gss_buffer_t tok;
{
     int len, ret;

     len = htonl((OM_uint32)tok->length);
     ret = write_all(s, (char *) &len, sizeof(int));
     if (ret < 0) {
          perror("sending token length");
          return -1;
     } else if (ret != 4) {
         if (display_file)
             fprintf(display_file,
                     "sending token length: %d of %d bytes written\n",
                     ret, 4);
          return -1;
     }

     ret = write_all(s, tok->value, (OM_uint32)tok->length);
     if (ret < 0) {
          perror("sending token data");
          return -1;
     } else if (ret != tok->length) {
         if (display_file)
             fprintf(display_file,
                     "sending token data: %d of %d bytes written\n",
                     ret, tok->length);
         return -1;
     }

     return 0;
}
```

recv_token()

This function receives a token or message.

EXAMPLE A–19 recv_token()

```
/*
 * Function: recv_token
 *
 * Purpose: Reads a token from a file descriptor.
 *
 * Arguments:
 *
 *      s               (r) an open file descriptor
 *      tok             (w) the read token
 *
```

EXAMPLE A–19 recv_token() *(Continued)*

```
 * Returns: 0 on success, -1 on failure
 *
 * Effects:
 *
 * recv_token reads the token length (as a network long), allocates
 * memory to hold the data, and then reads the token data from the
 * file descriptor s.  It blocks to read the length and data, if
 * necessary.  On a successful return, the token should be freed with
 * gss_release_buffer.  It returns 0 on success, and -1 if an error
 * occurs or if it could not read all the data.
 */
int recv_token(s, tok)
     int s;
     gss_buffer_t tok;
{
     int ret, len;

     ret = read_all(s, (char *) &len, sizeof(int));
     if (ret < 0) {
          perror("reading token length");
          return -1;
     } else if (ret != 4) {
         if (display_file)
             fprintf(display_file,
                     "reading token length: %d of %d bytes read\n",
                     ret, 4);
         return -1;
     }

     tok->length = ntohl(len);
     tok->value = (char *) malloc(tok->length);
     if (tok->value == NULL) {
         if (display_file)
             fprintf(display_file,
                     "Out of memory allocating token data\n");
          return -1;
     }

     ret = read_all(s, (char *) tok->value, (OM_uint32)tok->length);
     if (ret < 0) {
          perror("reading token data");
          free(tok->value);
          return -1;
     } else if (ret != tok->length) {
          fprintf(stderr, "sending token data: %d of %d bytes written\n",
                  ret, tok->length);
          free(tok->value);
          return -1;
     }

     return 0;
}
```

APPENDIX **B**

GSS-API Reference

This appendix includes the following sections:

- "GSS-API Functions" on page 115 provides a table of GSS-API functions.
- "GSS-API Status Codes" on page 118 discusses status codes returned by GSS-API functions, and provides a list of those status codes.
- "GSS-API Data Types and Values" on page 122 discusses the various data types used by the GSS-API.

Additional GSS-API definitions can be found in the file `gssapi.h`.

GSS-API Functions

The following table lists the functions of the GSS-API. For more information on each function, see its man page. See also "Functions From Previous Versions of the GSS-API" on page 117.

TABLE B–1 GSS-API Functions

Function	Description
`gss_acquire_cred()`	Assume a global identity; obtain a GSS-API credential handle for pre-existing credentials
`gss_add_cred()`	Construct credentials incrementally
`gss_inquire_cred()`	Obtain information about a credential
`gss_inquire_cred_by_mech()`	Obtain per-mechanism information about a credential

TABLE B–1 GSS-API Functions *(Continued)*

Function	Description
`gss_release_cred()`	Discard a credential handle
`gss_init_sec_context()`	Initiate a security context with a peer application
`gss_accept_sec_context()`	Accept a security context initiated by a peer application
`gss_delete_sec_context()`	Discard a security context
`gss_process_context_token()`	Process a token on a security context from a peer application
`gss_context_time()`	Determine for how long a context will remain valid
`gss_inquire_context()`	Obtain information about a security context
`gss_wrap_size_limit()`	Determine token-size limit for `gss_wrap()` on a context
`gss_export_sec_context()`	Transfer a security context to another process
`gss_import_sec_context()`	Import a transferred context
`gss_get_mic()`	Calculate a cryptographic message integrity code (MIC) for a message; integrity service
`gss_verify_mic()`	Check a MIC against a message; verify integrity of a received message
`gss_wrap()`	Attach a MIC to a message, and optionally encrypt the message content
`gss_unwrap()`	Verify a message with attached MIC, and decrypt message content if necessary
`gss_import_name()`	Convert a contiguous string name to internal-form
`gss_display_name()`	Convert internal-form name to text
`gss_compare_name()`	Compare two internal-form names
`gss_release_name()`	Discard an internal-form name
`gss_inquire_names_for_mech()`	List the name types supported by the specified mechanism
`gss_inquire_mechs_for_name()`	List mechanisms that support the specified name type
`gss_canonicalize_name()`	Convert an internal name to an MN

TABLE B–1 GSS-API Functions *(Continued)*

Function	Description
`gss_export_name()`	Convert an MN to export form
`gss_duplicate_name()`	Create a copy of an internal name
`gss_add_oid_set_member()`	Add an object identifier to a set
`gss_display_status()`	Convert a GSS-API status code to text
`gss_indicate_mechs()`	Determine available underlying authentication mechanisms
`gss_release_buffer()`	Discard a buffer
`gss_release_oid_set()`	Discard a set of object identifiers
`gss_create_empty_oid_set()`	Create a set containing no object identifiers
`gss_test_oid_set_member()`	Determine whether an object identifier is a member of a set

Functions From Previous Versions of the GSS-API

This section explains functions that were included in previous versions of the GSS-API.

Functions for Manipulating OIDs

The following functions are supported by the Sun implementation of the GSS-API for convenience and for backward compatibility with programs written for older versions of the GSS-API. However, they should not be relied upon, as they might not be supported by other implementations of the GSS-API.

- `gss_delete_oid()`
- `gss_oid_to_str()`
- `gss_str_to_oid()`

Although these functions make it possible to convert a mechanism's name from a string to an OID, programmers should use the default GSS-API mechanism, instead of specifying one, if at all possible.

Renamed Functions

The following functions have been supplanted by newer functions. In each case, the new function is the functional equivalent of the old one. Although the old functions are supported, developers should replace them with the newer functions whenever possible.

- `gss_sign()` has been replaced with `gss_get_mic()`.
- `gss_verify()` has been replaced with `gss_verify_mic()`.
- `gss_seal()` has been replaced with `gss_wrap()`.
- `gss_unseal()` has been replaced with `gss_unwrap()`.

GSS-API Status Codes

Major status codes are encoded in the `OM_uint32` as shown in Figure B–1.

Major Status Code OM_uint32

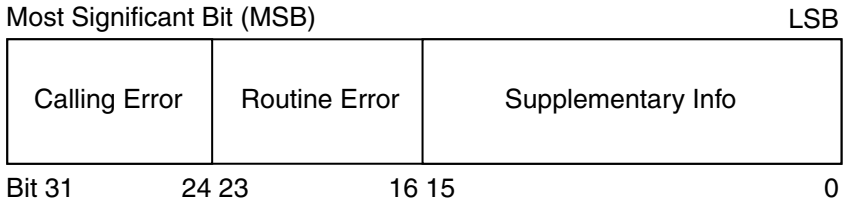

FIGURE B–1 Major-Status Encoding

If a GSS-API routine returns a GSS status code whose upper 16 bits contain a non-zero value, the call has failed. If the calling error field is non-zero, the invoking application's call of the routine was erroneous. *Calling errors* are listed in Table B–2. If the routine error field is non-zero, the routine failed because of a *routine-specific error*, as listed below in Table B–3. Whether or not the upper 16 bits indicate a failure or a success, the routine might indicate additional information by setting bits in the *supplementary information* field of the status code. The meaning of individual bits is listed in Table B–4.

GSS-API Major Status Code Values

The following tables lists calling errors returned by the GSS-API; that is, errors that are specific to a particular language-binding (C, in this case).

TABLE B–2 Calling Errors

Error	Value in Field	Meaning
GSS_S_CALL_INACCESSIBLE_READ	1	A required input parameter could not be read
GSS_S_CALL_INACCESSIBLE_WRITE	2	A required output parameter could not be written
GSS_S_CALL_BAD_STRUCTURE	3	A parameter was malformed

The following table lists the routine errors (that is, generic errors returned by GSS-API functions).

TABLE B–3 Routine Errors

Error	Value in Field	Meaning
GSS_S_BAD_MECH	1	An unsupported mechanism was requested
GSS_S_BAD_NAME	2	An invalid name was supplied
GSS_S_BAD_NAMETYPE	3	A supplied name was of an unsupported type
GSS_S_BAD_BINDINGS	4	Incorrect channel bindings were supplied
GSS_S_BAD_STATUS	5	An invalid status code was supplied
GSS_S_BAD_MIC, GSS_S_BAD_SIG	6	A token had an invalid MIC
GSS_S_NO_CRED	7	No credentials were supplied, or the credentials were unavailable or inaccessible
GSS_S_NO_CONTEXT	8	No context has been established
GSS_S_DEFECTIVE_TOKEN	9	A token was invalid
GSS_S_DEFECTIVE_CREDENTIAL	10	A credential was invalid
GSS_S_CREDENTIALS_EXPIRED	11	The referenced credentials have expired
GSS_S_CONTEXT_EXPIRED	12	The context has expired
GSS_S_FAILURE	13	Miscellaneous failure. The underlying mechanism detected an error for which no specific GSS–API status code is defined. The mechanism-specific status code (minor-status code) provides more details about the error.

TABLE B–3 Routine Errors *(Continued)*

Error	Value in Field	Meaning
GSS_S_BAD_QOP	14	The quality-of-protection requested could not be provided
GSS_S_UNAUTHORIZED	15	The operation is forbidden by local security policy
GSS_S_UNAVAILABLE	16	The operation or option is unavailable
GSS_S_DUPLICATE_ELEMENT	17	The requested credential element already exists
GSS_S_NAME_NOT_MN	18	The provided name was not a Mechanism Name (MN)

The routine documentation also uses the name `GSS_S_COMPLETE`, which is a zero value, to indicate an absence of any API errors or supplementary information bits.

The following table lists the supplementary information values returned by GSS-API functions.

TABLE B–4 Supplementary Information Codes

Code	Bit Number	Meaning
GSS_S_CONTINUE_NEEDED	0 (LSB)	Returned only by `gss_init_sec_context()` or `gss_accept_sec_context()`. The routine must be called again to complete its function
GSS_S_DUPLICATE_TOKEN	1	The token was a duplicate of an earlier token
GSS_S_OLD_TOKEN	2	The token's validity period has expired
GSS_S_UNSEQ_TOKEN	3	A later token has already been processed
GSS_S_GAP_TOKEN	4	An expected per-message token was not received

For more on status codes, see "Status Codes" on page 25.

Displaying Status Codes

The function `gss_display_status()` translates GSS-API status codes into text format, allowing them to be displayed to a user or put in a text log. Because `gss_display_status()` only displays one status code at a time, and some

functions can return multiple status conditions, it should be invoked as part of a loop. As long as `gss_display_status()` indicates a non-zero status code (in Example B–1, the value returned in the *message_context* parameter), another status code is available for the function to fetch.

EXAMPLE B–1 Displaying Status Codes with `gss_display_status()`

```
OM_uint32 message_context;
OM_uint32 status_code;
OM_uint32 maj_status;
OM_uint32 min_status;
gss_buffer_desc status_string;

...

message_context = 0;

do {

     maj_status = gss_display_status(
               &min_status,
               status_code,
               GSS_C_GSS_CODE,
               GSS_C_NO_OID,
               &message_context,
               &status_string);

     fprintf(stderr, "%.*s\n", \
               (int)status_string.length, \
               (char *)status_string.value);

     gss_release_buffer(&min_status, &status_string,);

} while (message_context != 0);
```

Status Code Macros

The macros `GSS_CALLING_ERROR()`, `GSS_ROUTINE_ERROR()` and `GSS_SUPPLEMENTARY_INFO()` are provided, each of which takes a GSS status code and removes all but the relevant field. For example, the value obtained by applying `GSS_ROUTINE_ERROR()` to a status code removes the calling errors and supplementary information fields, leaving only the routine errors field. The values delivered by these macros can be directly compared with a GSS_S_*xxx* symbol of the appropriate type. The macro `GSS_ERROR()` is also provided, which when applied to a GSS–API status code returns a non-zero value if the status code indicated a calling or routine error, and a zero value otherwise. All macros defined by the GSS-API evaluate their argument(s) exactly once.

GSS-API Data Types and Values

This section covers various types of GSS-API data types and values. Certain data types that are opaque to the user, such as `gss_cred_id_t` or `gss_name_t`, are not covered here, since there is no advantage to knowing their structure. This section explains the following:

- "Basic GSS-API Data Types" on page 122 — Shows the definitions of the OM_uint32, `gss_buffer_desc`, `gss_OID_desc`, `gss_OID_set_desc_struct`, and `gss_channel_bindings_struct` data types.
- "Name Types" on page 123 — Shows the various name formats recognized by the GSS-API for specifying names.
- "Address Types for Channel Bindings" on page 124 — Shows the various values that may be used as the *initiator_addrtype* and *acceptor_addrtype* fields of the `gss_channel_bindings_t` structure.

Basic GSS-API Data Types

These are some of the data types used by the GSS-API.

OM_uint32

The OM_uint32 is a platform-independent 32–bit unsigned integer.

`gss_buffer_desc`

This is the definition of the `gss_buffer_desc` and the gss_buffer_t pointer:

```
typedef struct gss_buffer_desc_struct {
        size_t length;
        void *value;
} gss_buffer_desc, *gss_buffer_t;
```

`gss_OID_desc`

This is the definition of the `gss_OID_desc` and the gss_OID pointer:

```
typedef struct gss_OID_desc_struct {
        OM_uint32 length;
        void*elements;
} gss_OID_desc, *gss_OID;
```

gss_OID_set_desc

This is the definition of the `gss_OID_set_desc` and the gss_OID_set pointer:

```
typedef struct gss_OID_set_desc_struct  {
        size_t  count;
        gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;
```

gss_channel_bindings_struct

This is the definition of the `gss_channel_bindings_struct` structure and the `gss_channel_bindings_t` pointer:

```
typedef struct gss_channel_bindings_struct {
        OM_uint32 initiator_addrtype;
        gss_buffer_desc initiator_address;
        OM_uint32 acceptor_addrtype;
        gss_buffer_desc acceptor_address;
        gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

Name Types

A name type indicates the format of the name with which it is associated. (See "Names" on page 17 and "OIDs" on page 24 for more on names and name types.) The GSS-API supports the following name types, which are all `gss_OID` types:

TABLE B–5 Name Types

Name Type	Meaning
GSS_C_NO_NAME	The recommended symbolic name GSS_C_NO_NAME indicates that no name is being passed within a particular value of a parameter used for the purpose of transferring names.
GSS_C_NO_OID	This value corresponds to a null input value instead of an actual object identifier. Where specified, it indicates interpretation of an associated name based on a mechanism-specific default printable syntax.

TABLE B–5 Name Types *(Continued)*

Name Type	Meaning
GSS_C_NT_ANONYMOUS	Provided as a means to identify anonymous names, and can be compared against in order to determine, in a mechanism-independent fashion, whether a name refers to an anonymous principal.
GSS_C_NT_EXPORT_NAME	A name that has been exported with the `gss_export_name()` function.
GSS_C_NT_HOSTBASED_SERVICE	This name type is used to represent services associated with host computers. This name form is constructed using two elements, "service" and "hostname," as follows: *service@hostname*.
GSS_C_NT_MACHINE_UID_NAME	This name type is used to indicate a numeric user identifier corresponding to a user on a local system. Its interpretation is OS-specific. The `gss_import_name()` function resolves this UID into a username, which is then treated as the User Name Form.
GSS_C_NT_STRING_STRING_UID_NAME	This name type is used to indicate a string of digits representing the numeric user identifier of a user on a local system. Its interpretation is OS-specific. This name type is similar to the Machine UID Form, except that the buffer contains a string representing the user ID.
GSS_C_NT_USER_NAME	A named user on a local system. Its interpretation is OS-specific. It takes the form: *username*.

Address Types for Channel Bindings

Table B–6 shows the possible values for the *initiator_addrtype* and *acceptor_addrtype* fields of the `gss_channel_bindings_struct` structure. These fields indicate the format that a name can take (for example, ARPAnet IMP address format or AppleTalk address format). Channel bindings are discussed in "Channel Bindings" on page 49.

TABLE B–6 Channel Binding Address Types

Field	Value (Decimal)	Address Type
GSS_C_AF_UNSPEC	0	Unspecified address type

TABLE B–6 Channel Binding Address Types *(Continued)*

Field	Value (Decimal)	Address Type
GSS_C_AF_LOCAL	1	Host-local
GSS_C_AF_INET	2	Internet address type (example: IP)
GSS_C_AF_IMPLINK	3	ARPAnet IMP
GSS_C_AF_PUP	4	pup protocols (example: BSP)
GSS_C_AF_CHAOS	5	MIT CHAOS protocol
GSS_C_AF_NS	6	XEROX NS
GSS_C_AF_NBS	7	nbs
GSS_C_AF_ECMA	8	ECMA
GSS_C_AF_DATAKIT	9	datakit protocols
GSS_C_AF_CCITT	10	CCITT
GSS_C_AF_SNA	11	IBM SNA
GSS_C_AF_DECnet	12	DECnet
GSS_C_AF_DLI	13	Direct data link interface
GSS_C_AF_LAT	14	LAT
GSS_C_AF_HYLINK	15	NSC Hyperchannel
GSS_C_AF_APPLETALK	16	AppleTalk
GSS_C_AF_BSC	17	BISYNC
GSS_C_AF_DSS	18	Distributed system services
GSS_C_AF_OSI	19	OSI TP4
GSS_C_AF_X25	21	X.25
GSS_C_AF_NULLADDR	255	No address specified

APPENDIX C

Specifying an OID

This chapter describes use of the default QOP and mechanism provided by the GSS-API.

Mechanisms and QOPs

Although it is strongly recommended that you use the default QOP and mechanism provided by the GSS-API if at all possible (see "OIDs" on page 24), you might have your own reasons for specifying these OIDs. For that reason this chapter briefly discusses how to specify OIDs.

Files Containing OID Values

For convenience, the GSS-API does allow mechanisms and QOPs to be displayed in human-readable form. On Solaris systems, two files, `/etc/gss/mech` and `/etc/gss/qop`, contain information about available mechanisms and QOPs. If you don't have access to these files (perhaps because a remote machine won't let you in), then you must provide the string literals from some other source, such as the published internet standard for that mechanism or QOP.

The `/etc/gss/mech` File

You can look in the `/etc/gss/mech` file to see which mechanisms are available; `/etc/gss/mech` contains their names in both numerical and alphabetic form. `/etc/gss/mech` presents the information in this format: the mechanism name, in ASCII; the mechanism's OID; the shared library implementing the services provided by this mechanism; and, optionally, the kernel module implementing the service. A sample `/etc/gss/mech` might look like Example C–1.

EXAMPLE C–1 The `/etc/gss/mech` File

```
#
# Copyright (c) 2000, by Sun Microsystems, Inc.
# All rights reserved.
#
#ident  "@(#)mech 1.6     00/12/04 SMI"
#
# This file contains the GSS-API based security mechanism names,
# its object identifier (OID) and a shared library that implements
# the services for that mechanism under GSS-API.
#
# Mechanism Name        Object Identifier       Shared Library  Kernel Module
#
diffie_hellman_640_0    1.3.6.4.1.42.2.26.2.4   dh640-0.so.1
diffie_hellman_1024_0   1.3.6.4.1.42.2.26.2.5   dh1024-0.so.1
kerberos_v5             1.2.840.113554.1.2.2    gl/mech_krb5.so gl_kmech_krb5
```

The `/etc/gss/qop` File

The `/etc/gss/qop` file stores, for all mechanisms installed, all the QOPs supported by each mechanism, both as an ASCII string as its corresponding 32–bit integer. A sample `/etc/gss/qop` might look like Example C–2.

EXAMPLE C–2 The `/etc/gss/qop` File

```
#
# Copyright (c) 2000, by Sun Microsystems, Inc.
# All rights reserved.
#
#ident  "@(#)qop 1.3     00/11/09 SMI"
#
# This file contains information about the GSS-API based quality of
# protection (QOP), its string name and its value (32-bit integer).
#
# QOP string                    QOP Value       Mechanism Name
#
GSS_KRB5_INTEG_C_QOP_DES_MD5    0               kerberos_v5
GSS_KRB5_CONF_C_QOP_DES         0               kerberos_v5
```

`gss_str_to_oid()`

For backward compatibility with earlier versions of the GSS-API, this implementation of the GSS-API supports the function `gss_str_to_oid()`. `gss_str_to_oid()` converts a string representing a mechanism or QOP (either as a number or a word) to an OID.

Caution – `gss_str_to_oid()`, `gss_oid_to_str()`, and `gss_release_oid()` are not supported by some implementations of the GSS-API in order to discourage the use of explicit, non-default mechanisms and QOPs.

The string can be hard-coded in the application, or come from user input. However, not all implementations of the GSS-API support this function, so applications shouldn't rely on it.

Note that the number representing a mechanism can have two different formats. The first, `{ 1 2 3 4 }`, is officially mandated by the GSS-API specifications, while the second, `1.2.3.4`, is more widely used but is not an official standard format. `gss_str_to_oid()` expects the mechanism number in the first format, so you must convert the string if it's in the second format before calling `gss_str_to_oid()`. An example of this is shown in "`parse_oid()`" on page 84. If the mechanism is not a valid one, `gss_str_to_oid()` returns `GSS_S_BAD_MECH`.

Because `gss_str_to_oid()` allocates GSS-API dataspace, the `gss_release_oid()` function exists, to remove the allocated OID when you've finished with it. Like `gss_str_to_oid()`, `gss_release_oid()` is not a generally supported function and should not be relied upon in programs that aspire to universal portability.

Constructing Mechanism OIDs

Since `gss_str_to_oid()` is not always available or desirable, there are preferable, if more complex, ways to find out which mechanisms are available, and to choose one. One way is to construct a mechanism OID "by hand" and then compare it to a set of available mechanisms; another way is to get the set of available mechanisms and choose one from it.

The `gss_OID` type has the following form:

```
typedef struct gss_OID_desc struct {
     OM_uint32 length;
     void          *elements;
} gss_OID_desc, *gss_OID;
```

where the *elements* field of this structure points to the first byte of an octet string containing the ASN.1 BER encoding of the value portion of the normal BER TLV encoding of the `gss_OID`. The *length* field contains the number of bytes in this value. For example, the `gss_OID` value corresponding to the DASS X.509 authentication mechanism, has a *length* field of 7 and an *elements* field pointing to seven octets containing the following octal values: 53,14,2,207,163,7,5.

One way to construct a mechanism OID is to declare a `gss_OID` and then initialize its elements "by hand" to represent that of a given mechanism. (As above, the input for the *elements* values might be hard-coded, be looked up in a table, or come from user input.) This is somewhat more painstaking than using `gss_str_to_oid()` but achieves the same effect.

Such a `gss_OID` can then be compared against a set of available mechanisms returned by the functions `gss_indicate_mechs()` or `gss_inquire_mechs_for_name()`. The application can check to see if its constructed mechanism OID is in this set of available mechanisms by using the `gss_test_oid_set_member()` function. If `gss_test_oid_set_member()` does not return an error, then the constructed OID can be used as the mechanism for GSS-API transactions.

As an alternative to constructing a pre-set OID, the application can use `gss_indicate_mechs()` or `gss_inquire_mechs_for_name()` to get the `gss_OID_set` of available mechanisms. A `gss_OID_set` has the following form:

```
typedef struct gss_OID_set_desc_struct {
     OM_uint32 length;
     void           *elements;
} gss_OID_set_desc, *gss_OID_set;
```

where each of the elements is a `gss_OID` representing a mechanism. The application can then parse each mechanism and display the element values of each one, in effect displaying the numerical representation of each mechanism. A user can then choose which of the mechanisms to use, based on this display, and the application then sets its mechanism to be the appropriate member of the `gss_OID_set`. Or the application can compare these desired mechanisms with a list of preferred mechanisms.

APPENDIX **D**

Sun-Specific Features

This appendix covers features unique to Sun's implementation of the GSS-API.

Implementation-Specifc Features

A few aspects of the GSS-API may differ from one implementation of the API to another. In most cases differences in implementations have only minimal effect on programs; in all cases developers can maximize portability by not relying on any behavior specific to a given implementation, including Sun's.

Sun-Specific Functions

There are no GSS-API functions that are unique to Sun's implementation.

Human-Readable Name Syntax

Implementations of the GSS-API may differ in the printable syntax that corresponds to names. Applications that aim for portability should refrain from comparing names based on human—readable (that is, printable) forms and should instead use `gss_compare_name()` to determine whether or not one internal-format name matches another.

Sun's implementation of `gss_display_name()` displays names as follows: if the *input_name*argument denotes a user principal, the `gss_display_name()` returns `user_principal@realm` as the *output_name_buffer*, and the `gss_OID` value as the *output_name_type*. If Kerberos v5 is the underlying mechanism, `gss_OID` is 1.2.840.11354.1.2.2.

If the name given to `gss_display_name()` was created by a call to `gss_import_name()`, specifying `GSS_C_NO_OID` as the name type, `gss_display_name()` returns `GSS_C_NO_OID` via the *output_name_type* parameter.

Format of Anonymous Names

The `gss_display_name()` function outputs the string '`<anonymous>`' to indicate an anonymous GSS-API principal. The name type OID associated with this name is GSS_C_NT_ANONYMOUS. No other valid printable names supported by Sun's implementation can begin with '<' and end with '>'.

Implementations of Selected Data Types

The following data types have been implemented as pointers (some implementations may specify them as arithmetic types): gss_cred_t, gss_ctx_id_t, gss_name_t.

Deletion of Contexts and Stored Data

In the case where context establishment fails, Sun's implementation does not automatically delete "half-built" contexts. Applications should therefore handle this event by deleting the contexts themselves with `gss_delete_sec_context()`.

Sun's implementation automatically releases stored data, such as internal names, through memory management. However, for good measure, applications should still call appropriate functions, such as `gss_release_name()`, when data elements are no longer needed.

Protection of Channel-Binding Information

Sun does not encrypt information contained in channel bindings. Programmers must therefore not assume that this information is unassailable.

Context Exportation and Interprocess Tokens

Sun's implementation supports context exportation; other implementations of the GSS-API may not. The interprocess token used in exporting a context may contain sensitive data from the original security context, including cryptographic keys. Sun's implementation of the GSS-API does *not* encrypt interprocess tokens. Therefore, applications that export security contexts must take appropriate steps to protect these tokens in transit (that is, wrap them themselves).

Sun's implementation detects and rejects attempted multiple imports of the same context.

Types of Credentials Supported

Sun's implementation of the GSS-API supports the acquisition of `GSS_C_INITIATE`, `GSS_C_ACCEPT`, and `GSS_C_BOTH` credentials via `gss_acquire_cred()`.

Credential Expiration

The Sun implementation of the GSS-API supports credential expiration. Therefore, programmers can use parameters relating to credential lifetime in functions such as `gss_acquire_cred()` and `gss_add_cred()`.

Context Expiration

Sun's implementation of the GSS-API supports context expiration. Therefore, programmers can use parameters relating to context lifetime in functions such as `gss_init_sec_context()` and `gss_inquire_context()`.

Wrap Size Limits and QOP Values

Sun's implementation of the GSS-API (as opposed to any underlying mechanism) does not impose a maximum size on the size of messages to be processed by `gss_wrap()`. Applications can determine the maximum message size with `gss_wrap_size_limit()`.

Sun's implementation of the GSS-API detects invalid QOP values when `gss_wrap_size_limit()` is called.

Use of *minor_status* Parameter

In Sun's implementation of the GSS-API, functions return only mechanism-specific information in the *minor_status* parameter. (Other implementations may include implementation-specific return values as part of the returned minor-status code.)

APPENDIX **E**

Kerberos v5 Status Codes

This chapter lists and describes the Kerberos v5 status codes.

Table of Kerberos v5 Status Codes

Each GSS-API function returns two status codes: a *major status code* and a *minor status code*. Major status codes relate to the behavior of the GSS-API itself. For example, if an application attempts to transmit a message after a security context has expired, the GSS-API returns a major status code of `GSS_S_CONTEXT_EXPIRED`. Major status codes are listed in "GSS-API Status Codes" on page 118.

Minor status codes are returned by the underlying security mechanisms supported by a given implementation of the GSS-API. At present, the only such mechanism supported by Sun's implementation of the GSS-API is Kerberos v5. (Sun's implementation of the Kerberos v5 is known as SEAM, the Sun Enterprise Authentication Mechanism; for our purposes, you can think of them as the same thing.) Every GSS-API function takes as its first argument a *minor_status* (or *minor_stat*) parameter; an application can examine this parameter when the function returns, successfully or not, to see what the status the underlying mechanism reports.

The following table lists the status messages that might be returned by Kerberos v5 in the *minor_status* argument.

For more on GSS-API status codes, see "Status Codes" on page 25.

TABLE E–1 Kerberos v5 Status Codes 1

Minor Status	Value	Meaning
KRB5KDC_ERR_NONE	-1765328384L	No error

TABLE E–1 Kerberos v5 Status Codes 1 *(Continued)*

Minor Status	Value	Meaning
KRB5KDC_ERR_NAME_EXP	-1765328383L	Client's entry in database has expired
KRB5KDC_ERR_SERVICE_EXP	-1765328382L	Server's entry in database has expired
KRB5KDC_ERR_BAD_PVNO	-1765328381L	Requested protocol version not supported
KRB5KDC_ERR_C_OLD_MAST_KVNO	-1765328380L	Client's key is encrypted in an old master key
KRB5KDC_ERR_S_OLD_MAST_KVNO	-1765328379L	Server's key is encrypted in an old master key
KRB5KDC_ERR_C_PRINCIPAL_UNKNOWN	-1765328378L	Client not found in Kerberos database
KRB5KDC_ERR_S_PRINCIPAL_UNKNOWN	-1765328377L	Server not found in Kerberos database
KRB5KDC_ERR_PRINCIPAL_NOT_UNIQUE	-1765328376L	Principal has multiple entries in Kerberos database
KRB5KDC_ERR_NULL_KEY	-1765328375L	Client or server has a null key
KRB5KDC_ERR_CANNOT_POSTDATE	-1765328374L	Ticket is ineligible for postdating
KRB5KDC_ERR_NEVER_VALID	-1765328373L	Requested effective lifetime is negative or too short
KRB5KDC_ERR_POLICY	-1765328372L	KDC policy rejects request
KRB5KDC_ERR_BADOPTION	-1765328371L	KDC can't fulfill requested option
KRB5KDC_ERR_ETYPE_NOSUPP	-1765328370L	KDC has no support for encryption type
KRB5KDC_ERR_SUMTYPE_NOSUPP	-1765328369L	KDC has no support for checksum type
KRB5KDC_ERR_PADATA_TYPE_NOSUPP	-1765328368L	KDC has no support for padata type
KRB5KDC_ERR_TRTYPE_NOSUPP	-1765328367L	KDC has no support for transited type

TABLE E–1 Kerberos v5 Status Codes 1 *(Continued)*

Minor Status	Value	Meaning
KRB5KDC_ERR_CLIENT_REVOKED	-1765328366L	Client's credentials have been revoked
KRB5KDC_ERR_SERVICE_REVOKED	-1765328365L	Credentials for server have been revoked

TABLE E–2 Kerberos v5 Status Codes 2

Minor Status	Value	Meaning
KRB5KDC_ERR_TGT_REVOKED	-1765328364L	TGT has been revoked
KRB5KDC_ERR_CLIENT_NOTYET	-1765328363L	Client not yet valid — try again later
KRB5KDC_ERR_SERVICE_NOTYET	-1765328362L	Server not yet valid — try again later
KRB5KDC_ERR_KEY_EXP	-1765328361L	Password has expired
KRB5KDC_ERR_PREAUTH_FAILED	-1765328360L	Preauthentication failed
KRB5KDC_ERR_PREAUTH_REQUIRED	-1765328359L	Additional pre-authentication required
KRB5KDC_ERR_SERVER_NOMATCH	-1765328358L	Requested server and ticket don't match
KRB5PLACEHOLD_27 *through* KRB5PLACEHOLD_30	-1765328357L *through* -1765328354L	KRB5 error codes 27 *through* 30 (reserved)
KRB5KRB_AP_ERR_BAD_INTEGRITY	-1765328353L	Decrypt integrity check failed
KRB5KRB_AP_ERR_TKT_EXPIRED	-1765328352L	Ticket expired
KRB5KRB_AP_ERR_TKT_NYV	-1765328351L	Ticket not yet valid
KRB5KRB_AP_ERR_REPEAT	-1765328350L	Request is a replay
KRB5KRB_AP_ERR_NOT_US	-1765328349L	The ticket isn't for us
KRB5KRB_AP_ERR_BADMATCH	-1765328348L	Ticket/authenticator don't match
KRB5KRB_AP_ERR_SKEW	-1765328347L	Clock skew too great
KRB5KRB_AP_ERR_BADADDR	-1765328346L	Incorrect net address
KRB5KRB_AP_ERR_BADVERSION	-1765328345L	Protocol version mismatch

TABLE E–2 Kerberos v5 Status Codes 2 *(Continued)*

Minor Status	Value	Meaning
KRB5KRB_AP_ERR_MSG_TYPE	-1765328344L	Invalid message type
KRB5KRB_AP_ERR_MODIFIED	-1765328343L	Message stream modified
KRB5KRB_AP_ERR_BADORDER	-1765328342L	Message out of order
KRB5KRB_AP_ERR_ILL_CR_TKT	-1765328341L	Illegal cross-realm ticket
KRB5KRB_AP_ERR_BADKEYVER	-1765328340L	Key version is not available

TABLE E–3 Kerberos v5 Status Codes 3

Minor Status	Value	Meaning
KRB5KRB_AP_ERR_NOKEY	-1765328339L	Service key not available
KRB5KRB_AP_ERR_MUT_FAIL	-1765328338L	Mutual authentication failed
KRB5KRB_AP_ERR_BADDIRECTION	-1765328337L	Incorrect message direction
KRB5KRB_AP_ERR_METHOD	-1765328336L	Alternative authentication method required
KRB5KRB_AP_ERR_BADSEQ	-1765328335L	Incorrect sequence number in message
KRB5KRB_AP_ERR_INAPP_CKSUM	-1765328334L	Inappropriate type of checksum in message
KRB5PLACEHOLD_51 *through* KRB5PLACEHOLD_59	-1765328333L *through* -1765328325L	KRB5 error codes 51 *through* 59 (reserved)
KRB5KRB_ERR_GENERIC	-1765328324L	Generic error
KRB5KRB_ERR_FIELD_TOOLONG	-1765328323L	Field is too long for this implementation
KRB5PLACEHOLD_62 *through* KRB5PLACEHOLD_127	-1765328322L *through* -1765328257L	KRB5 error codes 62 *through* 127 (reserved)
(value not returned)	-1765328256L	*For internal use only*
KRB5_LIBOS_BADLOCKFLAG	-1765328255L	Invalid flag for file lock mode
KRB5_LIBOS_CANTREADPWD	-1765328254L	Cannot read password

TABLE E–3 Kerberos v5 Status Codes 3 *(Continued)*

Minor Status	Value	Meaning
KRB5_LIBOS_BADPWDMATCH	-1765328253L	Password mismatch
KRB5_LIBOS_PWDINTR	-1765328252L	Password read interrupted
KRB5_PARSE_ILLCHAR	-1765328251L	Illegal character in component name
KRB5_PARSE_MALFORMED	-1765328250L	Malformed representation of principal
KRB5_CONFIG_CANTOPEN	-1765328249L	Can't open/find Kerberos `/etc/krb5/krb5` configuration file
KRB5_CONFIG_BADFORMAT	-1765328248L	Improper format of Kerberos `/etc/krb5/krb5` configuration file
KRB5_CONFIG_NOTENUFSPACE	-1765328247L	Insufficient space to return complete information
KRB5_BADMSGTYPE	-1765328246L	Invalid message type specified for encoding
KRB5_CC_BADNAME	-1765328245L	Credential cache name malformed

TABLE E–4 Kerberos v5 Status Codes 4

Minor Status	Value	Meaning
KRB5_CC_UNKNOWN_TYPE	-1765328244L	Unknown credential cache type
KRB5_CC_NOTFOUND	-1765328243L	Matching credential not found
KRB5_CC_END	-1765328242L	End of credential cache reached
KRB5_NO_TKT_SUPPLIED	-1765328241L	Request did not supply a ticket
KRB5KRB_AP_WRONG_PRINC	-1765328240L	Wrong principal in request

TABLE E–4 Kerberos v5 Status Codes 4 *(Continued)*

Minor Status	Value	Meaning
KRB5KRB_AP_ERR_TKT_INVALID	-1765328239L	Ticket has invalid flag set
KRB5_PRINC_NOMATCH	-1765328238L	Requested principal and ticket don't match
KRB5_KDCREP_MODIFIED	-1765328237L	KDC reply did not match expectations
KRB5_KDCREP_SKEW	-1765328236L	Clock skew too great in KDC reply
KRB5_IN_TKT_REALM_MISMATCH	-1765328235L	Client/server realm mismatch in initial ticket request
KRB5_PROG_ETYPE_NOSUPP	-1765328234L	Program lacks support for encryption type
KRB5_PROG_KEYTYPE_NOSUPP	-1765328233L	Program lacks support for key type
KRB5_WRONG_ETYPE	-1765328232L	Requested encryption type not used in message
KRB5_PROG_SUMTYPE_NOSUPP	-1765328231L	Program lacks support for checksum type
KRB5_REALM_UNKNOWN	-1765328230L	Cannot find KDC for requested realm
KRB5_SERVICE_UNKNOWN	-1765328229L	Kerberos service unknown
KRB5_KDC_UNREACH	-1765328228L	Cannot contact any KDC for requested realm
KRB5_NO_LOCALNAME	-1765328227L	No local name found for principal name
KRB5_MUTUAL_FAILED	-1765328226L	Mutual authentication failed
KRB5_RC_TYPE_EXISTS	-1765328225L	Replay cache type is already registered
KRB5_RC_MALLOC	-1765328224L	No more memory to allocate (in replay cache code)

TABLE E–4 Kerberos v5 Status Codes 4 *(Continued)*

Minor Status	Value	Meaning
KRB5_RC_TYPE_NOTFOUND	-1765328223L	Replay cache type is unknown

TABLE E–5 Kerberos v5 Status Codes 5

Minor Status	Value	Meaning
KRB5_RC_UNKNOWN	-1765328222L	Generic unknown RC error
KRB5_RC_REPLAY	-1765328221L	Message is a replay
KRB5_RC_IO	-1765328220L	Replay I/O operation failed
KRB5_RC_NOIO	-1765328219L	Replay cache type does not support non-volatile storage
KRB5_RC_PARSE	-1765328218L	Replay cache name parse/format error
KRB5_RC_IO_EOF	-1765328217L	End-of-file on replay cache I/O
KRB5_RC_IO_MALLOC	-1765328216L	No more memory to allocate (in replay cache I/O code)
KRB5_RC_IO_PERM	-1765328215L	Permission denied in replay cache code
KRB5_RC_IO_IO	-1765328214L	I/O error in replay cache i/o code
KRB5_RC_IO_UNKNOWN	-1765328213L	Generic unknown RC/IO error
KRB5_RC_IO_SPACE	-1765328212L	Insufficient system space to store replay information
KRB5_TRANS_CANTOPEN	-1765328211L	Can't open/find realm translation file
KRB5_TRANS_BADFORMAT	-1765328210L	Improper format of realm translation file
KRB5_LNAME_CANTOPEN	-1765328209L	Can't open/find lname translation database

TABLE E–5 Kerberos v5 Status Codes 5 *(Continued)*

Minor Status	Value	Meaning
KRB5_LNAME_NOTRANS	-1765328208L	No translation available for requested principal
KRB5_LNAME_BADFORMAT	-1765328207L	Improper format of translation database entry
KRB5_CRYPTO_INTERNAL	-1765328206L	Cryptosystem internal error
KRB5_KT_BADNAME	-1765328205L	Key table name malformed
KRB5_KT_UNKNOWN_TYPE	-1765328204L	Unknown Key table type
KRB5_KT_NOTFOUND	-1765328203L	Key table entry not found
KRB5_KT_END	-1765328202L	End of key table reached
KRB5_KT_NOWRITE	-1765328201L	Cannot write to specified key table

TABLE E–6 Kerberos v5 Status Codes 6

Minor Status	Value	Meaning
KRB5_KT_IOERR	-1765328200L	Error writing to key table
KRB5_NO_TKT_IN_RLM	-1765328199L	Cannot find ticket for requested realm
KRB5DES_BAD_KEYPAR	-1765328198L	DES key has bad parity
KRB5DES_WEAK_KEY	-1765328197L	DES key is a weak key
KRB5_BAD_ENCTYPE	-1765328196L	Bad encryption type
KRB5_BAD_KEYSIZE	-1765328195L	Key size is incompatible with encryption type
KRB5_BAD_MSIZE	-1765328194L	Message size is incompatible with encryption type
KRB5_CC_TYPE_EXISTS	-1765328193L	Credentials cache type is already registered
KRB5_KT_TYPE_EXISTS	-1765328192L	Key table type is already registered

TABLE E–6 Kerberos v5 Status Codes 6 *(Continued)*

Minor Status	Value	Meaning
KRB5_CC_IO	-1765328191L	Credentials cache I/O operation failed
KRB5_FCC_PERM	-1765328190L	Credentials cache file permissions incorrect
KRB5_FCC_NOFILE	-1765328189L	No credentials cache file found
KRB5_FCC_INTERNAL	-1765328188L	Internal file credentials cache error
KRB5_CC_WRITE	-1765328187L	Error writing to credentials cache file
KRB5_CC_NOMEM	-1765328186L	No more memory to allocate (in credentials cache code)
KRB5_CC_FORMAT	-1765328185L	Bad format in credentials cache
KRB5_INVALID_FLAGS	-1765328184L	Invalid KDC option combination (library internal error)
KRB5_NO_2ND_TKT	-1765328183L	Request missing second ticket
KRB5_NOCREDS_SUPPLIED	-1765328182L	No credentials supplied to library routine
KRB5_SENDAUTH_BADAUTHVERS	-1765328181L	Bad sendauth version was sent
KRB5_SENDAUTH_BADAPPLVERS	-1765328180L	Bad application version was sent (by sendauth)
KRB5_SENDAUTH_BADRESPONSE	-1765328179L	Bad response (during sendauth exchange)
KRB5_SENDAUTH_REJECTED	-1765328178L	Server rejected authentication (during sendauth exchange)

TABLE E–7 Kerberos v5 Status Codes 7

Minor Status	Value	Meaning
KRB5_PREAUTH_BAD_TYPE	-1765328177L	Unsupported pre-authentication type

TABLE E–7 Kerberos v5 Status Codes 7 *(Continued)*

Minor Status	Value	Meaning
KRB5_PREAUTH_NO_KEY	-1765328176L	Required pre-authentication key not supplied
KRB5_PREAUTH_FAILED	-1765328175L	Generic preauthentication failure
KRB5_RCACHE_BADVNO	-1765328174L	Unsupported replay cache format version number
KRB5_CCACHE_BADVNO	-1765328173L	Unsupported credentials cache format version number
KRB5_KEYTAB_BADVNO	-1765328172L	Unsupported key table format version number
KRB5_PROG_ATYPE_NOSUPP	-1765328171L	Program lacks support for address type
KRB5_RC_REQUIRED	-1765328170L	Message replay detection requires rcache parameter
KRB5_ERR_BAD_HOSTNAME	-1765328169L	Host name cannot be canonicalized
KRB5_ERR_HOST_REALM_UNKNOWN	-1765328168L	Cannot determine realm for host
KRB5_SNAME_UNSUPP_NAMETYPE	-1765328167L	Conversion to service principal undefined for name type
KRB5KRB_AP_ERR_V4_REPLY	-1765328166L	Initial Ticket response appears to be Version 4 error
KRB5_REALM_CANT_RESOLVE	-1765328165L	Cannot resolve KDC for requested realm
KRB5_TKT_NOT_FORWARDABLE	-1765328164L	Requesting ticket can't get forwardable tickets
KRB5_FWD_BAD_PRINCIPAL	-1765328163L	Bad principal name while trying to forward credentials
KRB5_GET_IN_TKT_LOOP	-1765328162L	Looping detected inside krb5_get_in_tkt

TABLE E–7 Kerberos v5 Status Codes 7 *(Continued)*

Minor Status	Value	Meaning
KRB5_CONFIG_NODEFREALM	-1765328161L	Configuration file `/etc/krb5/krb5.conf` does not specify default realm
KRB5_SAM_UNSUPPORTED	-1765328160L	Bad SAM flags in obtain_sam_padata
KRB5_KT_NAME_TOOLONG	-1765328159L	Keytab name too long
KRB5_KT_KVNONOTFOUND	-1765328158L	Key version number for principal in key table is incorrect
KRB5_CONF_NOT_CONFIGURED	-1765328157L	Kerberos `/etc/krb5/krb5.conf` configuration file not configured
gERROR_TABLE_BASE_krb5	-1765328384L	default

Glossary

ACL — See Access Control List (ACL).

Access Control List (ACL) — A file containing a list of principals with certain access permissions. Typically, a server consults an access control list to verify that a client has permission to use its services. Note that a principal authenticated by GSS-API can still be denied services if an ACL does not permit them.

authentication — A security service that verifies the claimed identity of a principal.

authorization — The process of determining whether a principal can use a service, which objects the principal is allowed to access, and the type of access allowed for each.

client — Narrowly, a process that makes use of a network service on behalf of a user; for example, an application that uses `rlogin`. In some cases, a server can itself be a client of some other server or service. Informally, a principal that makes use of a service.

confidentiality — A security service that encrypts data; confidentiality also includes integrity and authentication services. See also authentication, integrity, service.

context — A "state of trust" between two applications. When a context has successfully been established between two peers, the context acceptor is aware that the context initiator is who it claims to be, and can verify and decrypt messages sent to it. If the context includes mutual authentication, then initiator knows the acceptor's identity is valid and can also verify and/or decrypt messages from it.

context-level token — See token.

credential — An information package that identifies a principal; a principal's "identification badge," specifying who the principal is and, often, what privileges it has. Credentials are produced by security mechanisms.

credential cache A storage space (usually a file) containing credentials stored by a given mechanism.

data replay Data replay is said to occur when a single message in a message stream is received more than once. Many security mechanisms support data replay detection. Replay detection, if available, must be requested at context-establishment time.

data type (Also *data type*) The form that a given piece of data takes — for example, an int, a string, a gss_name_t structure, or a gss_OID_set structure.

delegation If permitted by the underlying security mechanism, a principal (generally the context initiator) can designate a peer principal (usually the context acceptor) as a proxy by *delegating* its credentials to it. The delegated credentials can be used by the recipient to make requests on behalf of the original principal, as might be the case when a principal uses `rlogin` from machine to machine to machine.

exported name A name that has been converted from the GSS-API internal-name format (specifically, a Mechanism Name) to the GSS-API Exported Name format by `gss_export_name()`. An exported name can be compared with names that are in non-GSS-API string format with `memcmp()`. See also Mechanism Name (MN), name.

flavor Historically, *security flavor* and *authentication flavor* were equivalent terms, as a flavor indicated a type of authentication (AUTH_UNIX, AUTH_DES, AUTH_KERB). RPCSEC_GSS is also a security flavor, even though it provides integrity and confidentiality services in addition to authentication.

GSS-API The Generic Security Service Application Programming Interface. A network layer providing support for various modular security services. GSS-API provides for security authentication, integrity, and confidentiality services, and allows maximum portability of applications with regard to security. See also authentication, confidentiality, integrity.

host A machine accessible over a network.

integrity A security service that, in addition to user authentication, provides proof of the validity of transmitted data through cryptographic tagging. See also authentication, confidentiality, Message Integrity Code (MIC).

mechanism A software package that specifies cryptographic techniques to achieve data authentication or confidentiality. Examples include Kerberos v5 and Diffie-Hellman public key.

Mechanism Name (MN) A special instance of a GSS-API internal-format name. A normal internal-format GSS-API name may contain several instances of a name, each in the format of an underlying mechanism; a Mechanism

Name, however, is unique to a particular mechanism. Mechanism Names are generated by `gss_canonicalize_name()`.

message Data in the form of a `gss_buffer_t` object sent from one GSS-API-based application to its peer. An example of a message is "`ls`" sent to a remote `ftp` server.

A message can contain more than just the user-provided data. For example, `gss_wrap()` takes an unwrapped message and produces a wrapped one to be sent; the wrapped message includes both the original message and an accompanying MIC. GSS-API-generated information that does not include a message is a *token* — see token for more.

Message Integrity Code (MIC) A cryptographic "tag" attached to transmitted data to ensure the data's validity. The recipient of the data generates its own MIC and compares it to the one that was sent; if they're equal, the message is valid. Some MICs, such as those generated by `gss_get_mic()`, are visible to the application, while others, such as those generated by `gss_wrap()` or `gss_init_sec_context()`, are not.

message–level token See token.

MIC See Message Integrity Code (MIC).

MN See Mechanism Name (MN).

mutual authentication When a context is established, a context initiator must authenticate itself to the context acceptor. In some cases the initiator might request that the acceptor authenticate itself back. If the acceptor does so, the two are said to be *mutually authenticated*.

name The name of a principal, such as "joe@machine." Names in the GSS-API are handled through the gss_name_t structure, which is opaque to applications. See also exported name, Mechanism Name (MN), name type, principal.

name type The particular form that a name is given in. Name types are stored as gss_OID types and are used to indicate the format used for a name. For example, the name "joe@machine" would have a name type of GSS_C_NT_HOSTBASED_SERVICE. See also exported name, Mechanism Name (MN), name.

opacity See opaque.

opaque A particular piece of data is said to be *opaque* if its value or format is not normally visible to functions that use it. For example, the *input_token* parameter to `gss_init_sec_context()` is opaque to the application, but significant to the GSS-API; similarly, the *input_message* parameter to `gss_wrap()` is opaque to the GSS-API but important to the application doing the wrapping.

out-of-sequence detection Many security mechanisms can detect if messages in a message stream are received out of their proper order. Message detection, if available, must be requested at context-establishment time.

per-message token See token.

principal A uniquely named client/user or server/service instance that participates in a network communication; GSS–API–based transactions involve interactions between principals. Examples of principal names include:

- `joe`
- `joe@machine`
- `nfs@machine`
- `123.45.678.9`
- `ftp://ftp.company.com`

See also name, name type.

privacy See confidentiality.

QOP See Quality of Protection (QOP).

Quality of Protection (QOP) A parameter used to select the cryptographic algorithms to be used in conjunction with the integrity or confidentiality service. With integrity, the QOP specifies the algorithm for producing a Message Integrity Code (MIC); with confidentiality, it specifies the algorithm for both the MIC and message encryption.

replay detection Many security mechanisms can detect if a message in a message stream has been incorrectly repeated. Message replay detection, if available, must be requested at context-establishment time.

security flavor See flavor.

security mechanism See mechanism.

security service See service.

server A principal that provides a resource to network clients. For example, if you `rlogin` to the machine `boston.eng.acme.com`, then that machine is the server providing the `rlogin` service.

service 1. (Also, *network service*) A resource provided to network clients; often provided by more than one server. For example, if you `rlogin` to the machine `boston.eng.acme.com`, then that machine is the server providing the `rlogin` service.

2. A *security service* can be either integrity or confidentiality, providing a level of protection beyond authentication. See also authentication, integrity and confidentiality.

token

A data packet in the form of a GSS-API `gss_buffer_t` structure. Tokens are produced by GSS-API functions for transfer to peer applications.

Tokens come in two types. *Context-level tokens* contain information used to establish or manage a security context. For example, `gss_init_sec_context()` bundles a context initiator's credential handle, the target machine's name, flags for various requested services, and more into a token to be sent to the context acceptor.

Message tokens (also known as *per-message tokens* or *message-level tokens*) contain information generated by a GSS-API function from messages to be sent to a peer application. For example, `gss_get_mic()` produces an identifying cryptographic tag for a given message and stores it in a token to be sent to a peer with the message. (Technically, a token is considered to be separate from a message, which is why `gss_wrap()` is said to produce an *output_message* and not an *output_token*.)

See also message.

Index

E

F

G

H

I