

Generic Security Service

for version 0.0.6, 12 August 2003

Simon Josefsson (bug-gss@josefsson.org)

This manual is for *Generic Security Service*, last updated 12 August 2003, for Version 0.0.6.
Copyright © 2003 Simon Josefsson.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections including “Criticism of GSS”, with the Front-Cover Texts being “A GNU Manual,” and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled “GNU Free Documentation License.”

(a) The FSF’s Back-Cover Text is: “You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.”

Table of Contents

1	Introduction	1
1.1	Getting Started	1
1.2	Features	1
1.3	Supported Platforms	1
1.4	Bug Reports	3
2	Preparation	4
2.1	Header	4
2.2	Initialization	4
2.3	Version Check	4
2.4	Building the source	5
2.5	Out of Memory handling	5
3	Standard GSS API	7
3.1	Error Handling	7
3.1.1	GSS status codes	7
3.1.2	Mechanism-specific status codes	9
3.2	Credential Management	9
3.3	Context-Level Routines	15
3.4	Per-Message Routines	26
3.5	Name Manipulation	28
3.6	Miscellaneous Routines	33
4	Extended GSS API	38
5	Acknowledgements	39
	Appendix A Criticism of GSS	40
	Concept Index	42
	API Index	43

1 Introduction

GSS is an implementation of the Generic Security Service Application Program Interface (GSS-API). GSS-API is used by network servers (e.g., IMAP, SMTP) to provide security services, e.g., authenticate clients against servers. GSS consists of a library and a manual.

GSS is developed for the GNU/Linux system, but runs on over 20 platforms including most major Unix platforms and Windows, and many kind of devices including iPAQ handhelds and S/390 mainframes.

GSS is licensed under the GNU Public License.

1.1 Getting Started

This manual documents the GSS programming interface. All functions and data types provided by the library are explained.

The reader is assumed to possess basic familiarity with GSS-API and network programming in C or C++. For general GSS-API information, and some programming examples, there is a guide available online at <http://docs.sun.com/db/doc/816-1331>.

This manual can be used in several ways. If read from the beginning to the end, it gives a good introduction into the library and how it can be used in an application. Forward references are included where necessary. Later on, the manual can be used as a reference manual to get just the information needed about any particular interface of the library. Experienced programmers might want to start looking at the examples at the end of the manual, and then only read up those parts of the interface which are unclear.

1.2 Features

GSS might have a couple of advantages over other libraries doing a similar job.

It's Free Software

Anybody can use, modify, and redistribute it under the terms of the GNU General Public License.

It's thread-safe

No global variables are used and multiple library handles and session handles may be used in parallel.

It's internationalized

It handles non-ASCII names and user visible strings used in the library (e.g., error messages) can be translated into the users' language.

It's portable

It should work on all Unix like operating systems, including Windows.

1.3 Supported Platforms

GSS has at some point in time been tested on the following platforms.

1. Debian GNU/Linux 3.0 (Woody)
GCC 2.95.4 and GNU Make. This is the main development platform. `alphaev67-unknown-linux-gnu`, `alphaev6-unknown-linux-gnu`, `arm-unknown-linux-gnu`, `hppa-unknown-linux-gnu`, `hppa64-unknown-linux-gnu`, `i686-pc-linux-gnu`, `ia64-unknown-linux-gnu`, `m68k-unknown-linux-gnu`, `mips-unknown-linux-gnu`, `mipsel-unknown-linux-gnu`, `powerpc-unknown-linux-gnu`, `s390-ibm-linux-gnu`, `sparc-unknown-linux-gnu`.
2. Debian GNU/Linux 2.1
GCC 2.95.1 and GNU Make. `armv4l-unknown-linux-gnu`.
3. Tru64 UNIX
Tru64 UNIX C compiler and Tru64 Make. `alphaev67-dec-osf5.1`, `alphaev68-dec-osf5.1`.
4. SuSE Linux 7.1
GCC 2.96 and GNU Make. `alphaev6-unknown-linux-gnu`, `alphaev67-unknown-linux-gnu`.
5. SuSE Linux 7.2a
GCC 3.0 and GNU Make. `ia64-unknown-linux-gnu`.
6. RedHat Linux 7.2
GCC 2.96 and GNU Make. `alphaev6-unknown-linux-gnu`, `alphaev67-unknown-linux-gnu`, `ia64-unknown-linux-gnu`.
7. RedHat Linux 8.0
GCC 3.2 and GNU Make. `i686-pc-linux-gnu`.
8. RedHat Advanced Server 2.1
GCC 2.96 and GNU Make. `i686-pc-linux-gnu`.
9. Slackware Linux 8.0.01
GCC 2.95.3 and GNU Make. `i686-pc-linux-gnu`.
10. Mandrake Linux 9.0
GCC 3.2 and GNU Make. `i686-pc-linux-gnu`.
11. IRIX 6.5
MIPS C compiler, IRIX Make. `mips-sgi-irix6.5`.
12. AIX 4.3.2
IBM C for AIX compiler, AIX Make. `rs6000-ibm-aix4.3.2.0`.
13. Microsoft Windows 2000 (Cygwin)
GCC 3.2, GNU make. `i686-pc-cygwin`.
14. HP-UX 11
HP-UX C compiler and HP Make. `ia64-hp-hpux11.22`, `hppa2.0w-hp-hpux11.11`.
15. SUN Solaris 2.8
Sun WorkShop Compiler C 6.0 and SUN Make. `sparc-sun-solaris2.8`.

16. NetBSD 1.6
GCC 2.95.3 and GNU Make. `alpha-unknown-netbsd1.6`, `i386-unknown-netbsdelf1.6`.
17. OpenBSD 3.1 and 3.2
GCC 2.95.3 and GNU Make. `alpha-unknown-openbsd3.1`, `i386-unknown-openbsd3.1`.
18. FreeBSD 4.7
GCC 2.95.4 and GNU Make. `alpha-unknown-freebsd4.7`, `i386-unknown-freebsd4.7`.

If you use GSS on, or port GSS to, a new platform please report it to the author.

1.4 Bug Reports

If you think you have found a bug in GSS, please investigate it and report it.

- Please make sure that the bug is really in GSS, and preferably also check that it hasn't already been fixed in the latest version.
- You have to send us a test case that makes it possible for us to reproduce the bug.
- You also have to explain what is wrong; if you get a crash, or if the results printed are not good and in that case, in what way. Make sure that the bug report includes all information you would need to fix this kind of bug for someone else.

Please make an effort to produce a self-contained report, with something definite that can be tested or debugged. Vague queries or piecemeal messages are difficult to act on and don't help the development effort.

If your bug report is good, we will do our best to help you to get a corrected version of the software; if the bug report is poor, we won't do anything about it (apart from asking you to send better bug reports).

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please also send a note.

Send your bug report to:

`'bug-gss@josefsson.org'`

2 Preparation

To use GSS, you have to perform some changes to your sources and the build system. The necessary changes are small and explained in the following sections. At the end of this chapter, it is described how the library is initialized, and how the requirements of the library are verified.

A faster way to find out how to adapt your application for use with GSS may be to look at the examples at the end of this manual.

2.1 Header

All standard interfaces (data types and functions) of the official GSS API are defined in the header file `'gss/api.h'`. The file is taken verbatim from the RFC (after correcting a few typos) where it is known as `'gssapi.h'`. However, to be able to co-exist gracefully with other GSS-API implementation, the name `'gssapi.h'` was changed.

The header file `'gss.h'` includes `'gss/api.h'`, add a few non-standard extensions (by including `'gss/ext.h'`), takes care of including header files related to all supported mechanisms (e.g., `'gss/krb5.h'`) and finally add C++ namespace protection of all definitions. Therefore, including `'gss.h'` in your project is recommended over `'gss/api.h'`. If using `'gss.h'` instead of `'gss/api.h'` causes problems, it should be regarded a bug.

You must include either file in all programs using the library, either directly or through some other header file, like this:

```
#include <gss.h>
```

The name space of GSS is `gss_*` for function names, `gss_*` for data types and `GSS_*` for other symbols. In addition the same name prefixes with one prepended underscore are reserved for internal use and should never be used by an application.

Each supported GSS mechanism may want to expose mechanism specific functionality, and can do so through one or more header files under the `'gss/'` directory. The Kerberos 5 mechanism uses the file `'gss/krb5.h'`, but again, it is included (with C++ namespace fixes) from `'gss.h'`.

2.2 Initialization

GSS does not need to be initialized before it can be used.

In order to take advantage of the internationalisation features in GSS, e.g. translated error messages, the application must set the current locale using `setlocale()` before calling, e.g., `gss_display_status()`. This is typically done in `main()` as in the following example.

```
#include <gss.h>
#include <locale.h>
...
setlocale (LC_ALL, "");
```

2.3 Version Check

It is often desirable to check that the version of GSS used is indeed one which fits all requirements. Even with binary compatibility new features may have been introduced but due to problem with the dynamic linker an old version is actually used. So you may want to check that the version is okay right after program startup. The function is called `gss_check_version()` and is described formally in See [Chapter 4 \[Extended GSS API\], page 38](#).

The normal way to use the function is to put something similar to the following early in your `main()`:

```
#include <gss.h>
...
if (!gss_check_version (GSS_VERSION))
{
    printf ("gss_check_version() failed:\n"
           "Header file incompatible with shared library.\n");
    exit(1);
}
```

2.4 Building the source

If you want to compile a source file that includes the `'gss.h'` header file, you must make sure that the compiler can find it in the directory hierarchy. This is accomplished by adding the path to the directory in which the header file is located to the compilers include file search path (via the `'-I'` option).

However, the path to the include file is determined at the time the source is configured. To solve this problem, GSS uses the external package `pkg-config` that knows the path to the include file and other configuration options. The options that need to be added to the compiler invocation at compile time are output by the `'--cflags'` option to `pkg-config gss`. The following example shows how it can be used at the command line:

```
gcc -c foo.c `pkg-config gss --cflags`
```

Adding the output of `'pkg-config gss --cflags'` to the compilers command line will ensure that the compiler can find the `'gss.h'` header file.

A similar problem occurs when linking the program with the library. Again, the compiler has to find the library files. For this to work, the path to the library files has to be added to the library search path (via the `'-L'` option). For this, the option `'--libs'` to `pkg-config gss` can be used. For convenience, this option also outputs all other options that are required to link the program with the GSS library (for instance, the `'-lshishi'` option). The example shows how to link `'foo.o'` with GSS into a program `foo`.

```
gcc -o foo foo.o `pkg-config gss --libs`
```

Of course you can also combine both examples to a single command by specifying both options to `pkg-config`:

```
gcc -o foo foo.c `pkg-config gss --cflags --libs`
```

2.5 Out of Memory handling

The GSS API does not have a standard error code for the out of memory error condition. Instead of adding a non-standard error code, this library has chosen to adopt a different strategy. Out of memory handling happens in rare situations, but performing the out of memory error handling after almost all API function invocations pollute your source code and might make it harder to spot more serious problems. The strategy chosen improve code readability and robustness.

For most applications, aborting the application with an error message when the out of memory situation occur is the best that can be wished for. This is how the library behaves by default.

However, we realize that some applications may not want to have the GSS library abort execution in any situation. The GSS library support a hook to let the application regain control and perform its own cleanups when an out of memory situation has occurred. The application can define a function (having a void prototype, i.e., no return value and no parameters) and set the library variable `xalloc_fail_func` to that function. The variable should be declared as follows.

```
extern void (*xalloc_fail_func) (void);
```

The GSS library will invoke this function if an out of memory error occurs. Note that after this the GSS library is in an undefined state, so you must unload or restart the application to continue call GSS library functions. The hook is only intended to allow the application to log the situation in a special way. Of course, care must be taken to not allocate more memory, as that will likely also fail.

3 Standard GSS API

3.1 Error Handling

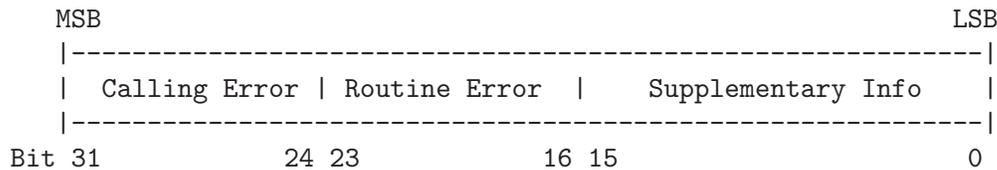
Every GSS-API routine returns two distinct values to report status information to the caller: GSS status codes and Mechanism status codes.

3.1.1 GSS status codes

GSS-API routines return GSS status codes as their `OM_uint32` function value. These codes indicate errors that are independent of the underlying mechanism(s) used to provide the security service. The errors that can be indicated via a GSS status code are either generic API routine errors (errors that are defined in the GSS-API specification) or calling errors (errors that are specific to these language bindings).

A GSS status code can indicate a single fatal generic API error from the routine and a single calling error. In addition, supplementary status information may be indicated via the setting of bits in the supplementary info field of a GSS status code.

These errors are encoded into the 32-bit GSS status code as follows:



Hence if a GSS-API routine returns a GSS status code whose upper 16 bits contain a non-zero value, the call failed. If the calling error field is non-zero, the invoking application's call of the routine was erroneous. Calling errors are defined in table 3-1. If the routine error field is non-zero, the routine failed for one of the routine-specific reasons listed below in table 3-2. Whether or not the upper 16 bits indicate a failure or a success, the routine may indicate additional information by setting bits in the supplementary info field of the status code. The meaning of individual bits is listed below in table 3-3.

Table 3-1 Calling Errors

Name	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

Table 3-2 Routine Errors

Name	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_NAME_TYPE	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 (see text)	
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	

Table 3-3 Supplementary Status Bits

Name	Bit Number	Meaning
----	-----	-----
GSS_S_CONTINUE_NEEDED	0 (LSB)	Returned only by <code>gss_init_sec_context</code> or <code>gss_accept_sec_context</code> . The routine must be called again to complete its function. See routine documentation for detailed description

<code>GSS_S_DUPLICATE_TOKEN</code>	1	The token was a duplicate of an earlier token
<code>GSS_S_OLD_TOKEN</code>	2	The token's validity period has expired
<code>GSS_S_UNSEQ_TOKEN</code>	3	A later token has already been processed
<code>GSS_S_GAP_TOKEN</code>	4	An expected per-message token was not received

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.

All `GSS_S_xxx` symbols equate to complete `OM_uint32` status codes, rather than to bitfield values. For example, the actual value of the symbol `GSS_S_BAD_NAME_TYPE` (value 3 in the routine error field) is $3 \ll 16$. 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 info fields, leaving only the routine errors field. The values delivered by these macros may 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 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 GSS-API evaluate their argument(s) exactly once.

A GSS-API implementation may choose to signal calling errors in a platform-specific manner instead of, or in addition to the routine value; routine errors and supplementary info should be returned via major status values only.

The GSS major status code `GSS_S_FAILURE` is used to indicate that the underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code will provide more details about the error.

3.1.2 Mechanism-specific status codes

GSS-API routines return a `minor_status` parameter, which is used to indicate specialized errors from the underlying security mechanism. This parameter may contain a single mechanism-specific error, indicated by a `OM_uint32` value.

The `minor_status` parameter will always be set by a GSS-API routine, even if it returns a calling error or one of the generic API errors indicated above as fatal, although most other output parameters may remain unset in such cases. However, output parameters that are expected to return pointers to storage allocated by a routine must always be set by the routine, even in the event of an error, although in such cases the GSS-API routine may elect to set the returned parameter value to `NULL` to indicate that no storage was actually allocated. Any length field associated with such pointers (as in a `gss_buffer_desc` structure) should also be set to zero in such cases.

3.2 Credential Management

Table 2-1 GSS-API Credential-management Routines

Routine -----	Section -----	Function -----
<code>gss_acquire_cred</code>	5.2	Assume a global identity; Obtain a GSS-API credential handle for pre-existing credentials.
<code>gss_add_cred</code>	5.3	Construct credentials incrementally
<code>gss_inquire_cred</code>	5.21	Obtain information about a credential
<code>gss_inquire_cred_by_mech</code>	5.22	Obtain per-mechanism information about a credential.
<code>gss_release_cred</code>	5.27	Discard a credential handle.

OM_uint32 `gss_acquire_cred` (OM_uint32 *minor_status, const [Function]
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: Integer, modify. Mechanism specific status code.

desired_name: gss_name_t, read. Name of principal whose credential should be acquired.

time_req: Integer, read, optional number of seconds that credentials should remain valid. Specify GSS_C_INDEFINITE to request that the credentials have the maximum permitted lifetime.

desired_mechs: Set of Object IDs, read, optional set of underlying security mechanisms that may be used. GSS_C_NO_OID_SET may be used to obtain an implementation-specific default.

cred_usage: gss_cred_usage_t, read. GSS_C_BOTH - Credentials may be used either to initiate or accept security contexts. GSS_C_INITIATE - Credentials will only be used to initiate security contexts. GSS_C_ACCEPT - Credentials will only be used to accept security contexts.

output_cred_handle: gss_cred_id_t, modify. The returned credential handle. Resources associated with this credential handle must be released by the application after use with a call to `gss_release_cred()`.

actual_mechs: Set of Object IDs, modify, optional. The set of mechanisms for which the credential is valid. Storage associated with the returned OID-set must be released by the application after use with a call to `gss_release_oid_set()`. Specify NULL if not required.

time_rec: Integer, modify, optional. Actual number of seconds for which the returned credentials will remain valid. If the implementation does not support expiration of credentials, the value GSS_C_INDEFINITE will be returned. Specify NULL if not required

Allows an application to acquire a handle for a pre-existing credential by name. GSS-API implementations must impose a local access-control policy on callers of this

routine to prevent unauthorized callers from acquiring credentials to which they are not entitled. This routine is not intended to provide a "login to the network" function, as such a function would involve the creation of new credentials rather than merely acquiring a handle to existing credentials. Such functions, if required, should be defined in implementation-specific extensions to the API.

If `desired_name` is `GSS_C_NO_NAME`, the call is interpreted as a request for a credential handle that will invoke default behavior when passed to `gss_init_sec_context()` (if `cred_usage` is `GSS_C_INITIATE` or `GSS_C_BOTH`) or `gss_accept_sec_context()` (if `cred_usage` is `GSS_C_ACCEPT` or `GSS_C_BOTH`).

Mechanisms should honor the `desired_mechs` parameter, and return a credential that is suitable to use only with the requested mechanisms. An exception to this is the case where one underlying credential element can be shared by multiple mechanisms; in this case it is permissible for an implementation to indicate all mechanisms with which the credential element may be used. If `desired_mechs` is an empty set, behavior is undefined.

This routine is expected to be used primarily by context acceptors, since implementations are likely to provide mechanism-specific ways of obtaining GSS-API initiator credentials from the system login process. Some implementations may therefore not support the acquisition of `GSS_C_INITIATE` or `GSS_C_BOTH` credentials via `gss_acquire_cred` for any name other than `GSS_C_NO_NAME`, or a name produced by applying either `gss_inquire_cred` to a valid credential, or `gss_inquire_context` to an active context.

If credential acquisition is time-consuming for a mechanism, the mechanism may choose to delay the actual acquisition until the credential is required (e.g. by `gss_init_sec_context` or `gss_accept_sec_context`). Such mechanism-specific implementation decisions should be invisible to the calling application; thus a call of `gss_inquire_cred` immediately following the call of `gss_acquire_cred` must return valid credential data, and may therefore incur the overhead of a deferred credential acquisition.

Valid return values and their meaning:

`GSS_S_COMPLETE`: Successful completion.

`GSS_S_BAD_MECH`: Unavailable mechanism requested.

`GSS_S_BAD_NAME_TYPE`: Type contained within `desired_name` parameter is not supported.

`GSS_S_BAD_NAME`: Value supplied for `desired_name` parameter is ill formed.

`GSS_S_CREDENTIALS_EXPIRED`: The credentials could not be acquired Because they have expired.

`GSS_S_NO_CRED`: No credentials were found for the specified name.

```
OM_uint32 gss_add_cred (OM_uint32 *minor_status, const [Function]
    gss_cred_id_t input_cred_handle, const gss_name_t desired_name,
    const gss_OID desired_mech, gss_cred_usage_t cred_usage, OM_uint32
    initiator_time_req, OM_uint32 acceptor_time_req, gss_cred_id_t
    *output_cred_handle, gss_OID_set *actual_mechs, OM_uint32
    *initiator_time_rec, OM_uint32 *acceptor_time_rec)
```

minor_status: Integer, modify. Mechanism specific status code.

input_cred_handle: `gss_cred_id_t`, read, optional. The credential to which a credential-element will be added. If `GSS_C_NO_CREDENTIAL` is specified, the routine will compose the new credential based on default behavior (see description above). Note that, while the credential-handle is not modified by `gss_add_cred()`, the underlying credential will be modified if `output_cred_handle` is `NULL`.

desired_name: `gss_name_t`, read. Name of principal whose credential should be acquired.

desired_mech: Object ID, read. Underlying security mechanism with which the credential may be used.

cred_usage: `gss_cred_usage_t`, read. `GSS_C_BOTH` - Credential may be used either to initiate or accept security contexts. `GSS_C_INITIATE` - Credential will only be used to initiate security contexts. `GSS_C_ACCEPT` - Credential will only be used to accept security contexts.

initiator_time_req: Integer, read, optional. number of seconds that the credential should remain valid for initiating security contexts. This argument is ignored if the composed credentials are of type `GSS_C_ACCEPT`. Specify `GSS_C_INDEFINITE` to request that the credentials have the maximum permitted initiator lifetime.

acceptor_time_req: Integer, read, optional. number of seconds that the credential should remain valid for accepting security contexts. This argument is ignored if the composed credentials are of type `GSS_C_INITIATE`. Specify `GSS_C_INDEFINITE` to request that the credentials have the maximum permitted initiator lifetime.

output_cred_handle: `gss_cred_id_t`, modify, optional. The returned credential handle, containing the new credential-element and all the credential-elements from `input_cred_handle`. If a valid pointer to a `gss_cred_id_t` is supplied for this parameter, `gss_add_cred` creates a new credential handle containing all credential-elements from the `input_cred_handle` and the newly acquired credential-element; if `NULL` is specified for this parameter, the newly acquired credential-element will be added to the credential identified by `input_cred_handle`.

The resources associated with any credential handle returned via this parameter must be released by the application after use with a call to `gss_release_cred()`.

actual_mechs: Set of Object IDs, modify, optional. The complete set of mechanisms for which the new credential is valid. Storage for the returned OID-set must be freed by the application after use with a call to `gss_release_oid_set()`. Specify `NULL` if not required.

initiator_time_rec: Integer, modify, optional. Actual number of seconds for which the returned credentials will remain valid for initiating contexts using the specified mechanism. If the implementation or mechanism does not support expiration of

credentials, the value `GSS_C_INDEFINITE` will be returned. Specify `NULL` if not required

acceptor_time_rec: Integer, modify, optional. Actual number of seconds for which the returned credentials will remain valid for accepting security contexts using the specified mechanism. If the implementation or mechanism does not support expiration of credentials, the value `GSS_C_INDEFINITE` will be returned. Specify `NULL` if not required

Adds a credential-element to a credential. The credential-element is identified by the name of the principal to which it refers. GSS-API implementations must impose a local access-control policy on callers of this routine to prevent unauthorized callers from acquiring credential-elements to which they are not entitled. This routine is not intended to provide a "login to the network" function, as such a function would involve the creation of new mechanism-specific authentication data, rather than merely acquiring a GSS-API handle to existing data. Such functions, if required, should be defined in implementation-specific extensions to the API.

If *desired_name* is `GSS_C_NO_NAME`, the call is interpreted as a request to add a credential element that will invoke default behavior when passed to `gss_init_sec_context()` (if *cred_usage* is `GSS_C_INITIATE` or `GSS_C_BOTH`) or `gss_accept_sec_context()` (if *cred_usage* is `GSS_C_ACCEPT` or `GSS_C_BOTH`).

This routine is expected to be used primarily by context acceptors, since implementations are likely to provide mechanism-specific ways of obtaining GSS-API initiator credentials from the system login process. Some implementations may therefore not support the acquisition of `GSS_C_INITIATE` or `GSS_C_BOTH` credentials via `gss_acquire_cred` for any name other than `GSS_C_NO_NAME`, or a name produced by applying either `gss_inquire_cred` to a valid credential, or `gss_inquire_context` to an active context.

If credential acquisition is time-consuming for a mechanism, the mechanism may choose to delay the actual acquisition until the credential is required (e.g. by `gss_init_sec_context` or `gss_accept_sec_context`). Such mechanism-specific implementation decisions should be invisible to the calling application; thus a call of `gss_inquire_cred` immediately following the call of `gss_add_cred` must return valid credential data, and may therefore incur the overhead of a deferred credential acquisition.

This routine can be used to either compose a new credential containing all credential-elements of the original in addition to the newly-acquire credential-element, or to add the new credential- element to an existing credential. If `NULL` is specified for the *output_cred_handle* parameter argument, the new credential-element will be added to the credential identified by *input_cred_handle*; if a valid pointer is specified for the *output_cred_handle* parameter, a new credential handle will be created.

If `GSS_C_NO_CREDENTIAL` is specified as the *input_cred_handle*, `gss_add_cred` will compose a credential (and set the *output_cred_handle* parameter accordingly) based on default behavior. That is, the call will have the same effect as if the application had first made a call to `gss_acquire_cred()`, specifying the same usage and passing `GSS_C_NO_NAME` as the *desired_name* parameter to obtain an explicit credential

handle embodying default behavior, passed this credential handle to `gss_add_cred()`, and finally called `gss_release_cred()` on the first credential handle.

If `GSS_C_NO_CREDENTIAL` is specified as the `input_cred_handle` parameter, a non-NULL `output_cred_handle` must be supplied.

Valid return values and their meaning:

`GSS_S_COMPLETE`: Successful completion.

`GSS_S_BAD_MECH`: Unavailable mechanism requested.

`GSS_S_BAD_NAME_TYPE`: Type contained within `desired_name` parameter is not supported.

`GSS_S_BAD_NAME`: Value supplied for `desired_name` parameter is ill-formed.

`GSS_S_DUPLICATE_ELEMENT`: The credential already contains an element for the requested mechanism with overlapping usage and validity period.

`GSS_S_CREDENTIALS_EXPIRED`: The required credentials could not be added because they have expired.

`GSS_S_NO_CRED`: No credentials were found for the specified name.

`OM_uint32 gss_inquire_cred` (`OM_uint32 *minor_status`, `const` [Function]
`gss_cred_id_t cred_handle`, `gss_name_t *name`, `OM_uint32 *lifetime`,
`gss_cred_usage_t *cred_usage`, `gss_OID_set *mechanisms`)

minor_status: Integer, modify. Mechanism specific status code

cred_handle: `gss_cred_id_t`, read. A handle that refers to the target credential. Specify `GSS_C_NO_CREDENTIAL` to inquire about the default initiator principal.

name: `gss_name_t`, modify, optional. The name whose identity the credential asserts. Storage associated with this name should be freed by the application after use with a call to `gss_release_name()`. Specify `NULL` if not required.

lifetime: Integer, modify, optional. The number of seconds for which the credential will remain valid. If the credential has expired, this parameter will be set to zero. If the implementation does not support credential expiration, the value `GSS_C_INDEFINITE` will be returned. Specify `NULL` if not required.

cred_usage: `gss_cred_usage_t`, modify, optional. How the credential may be used. One of the following: `GSS_C_INITIATE`, `GSS_C_ACCEPT`, `GSS_C_BOTH`. Specify `NULL` if not required.

mechanisms: `gss_OID_set`, modify, optional. Set of mechanisms supported by the credential. Storage associated with this OID set must be freed by the application after use with a call to `gss_release_oid_set()`. Specify `NULL` if not required.

Obtains information about a credential.

Valid return values and their meaning:

`GSS_S_COMPLETE`: Successful completion

`GSS_S_NO_CRED`: The referenced credentials could not be accessed.

`GSS_S_DEFECTIVE_CREDENTIAL`: The referenced credentials were invalid.

`GSS_S_CREDENTIALS_EXPIRED`: The referenced credentials have expired. If the lifetime parameter was not passed as `NULL`, it will be set to 0.

OM_uint32 gss_inquire_cred_by_mech (OM_uint32 *minor_status, [Function]
 const gss_cred_id_t cred_handle, const gss_OID mech_type, gss_name_t
 *name, OM_uint32 *initiator_lifetime, OM_uint32 *acceptor_lifetime,
 gss_cred_usage_t *cred_usage)

minor_status: Integer, modify. Mechanism specific status code

cred_handle: gss_cred_id_t, read. A handle that refers to the target credential. Specify GSS_C_NO_CREDENTIAL to inquire about the default initiator principal.

mech_type: gss_OID, read. The mechanism for which information should be returned.

name: gss_name_t, modify, optional. The name whose identity the credential asserts. Storage associated with this name must be freed by the application after use with a call to gss_release_name(). Specify NULL if not required.

initiator_lifetime: Integer, modify, optional. The number of seconds for which the credential will remain capable of initiating security contexts under the specified mechanism. If the credential can no longer be used to initiate contexts, or if the credential usage for this mechanism is GSS_C_ACCEPT, this parameter will be set to zero. If the implementation does not support expiration of initiator credentials, the value GSS_C_INDEFINITE will be returned. Specify NULL if not required.

acceptor_lifetime: Integer, modify, optional. The number of seconds for which the credential will remain capable of accepting security contexts under the specified mechanism. If the credential can no longer be used to accept contexts, or if the credential usage for this mechanism is GSS_C_INITIATE, this parameter will be set to zero. If the implementation does not support expiration of acceptor credentials, the value GSS_C_INDEFINITE will be returned. Specify NULL if not required.

cred_usage: gss_cred_usage_t, modify, optional. How the credential may be used with the specified mechanism. One of the following: GSS_C_INITIATE, GSS_C_ACCEPT, GSS_C_BOTH. Specify NULL if not required.

Obtains per-mechanism information about a credential.

Valid return values and their meaning:

GSS_S_COMPLETE: Successful completion

GSS_S_NO_CRED: The referenced credentials could not be accessed.

GSS_S_DEFECTIVE_CREDENTIAL: The referenced credentials were invalid.

GSS_S_CREDENTIALS_EXPIRED: The referenced credentials have expired. If the lifetime parameter was not passed as NULL, it will be set to 0.

OM_uint32 gss_release_cred (OM_uint32 *minor_status, [Function]
 gss_cred_id_t * cred_handle)

minor_status: Mechanism specific status code.

cred_handle: Optional opaque handle identifying credential to be released. If GSS_C_NO_CREDENTIAL is supplied, the routine will complete successfully, but will do nothing.

Informs GSS-API that the specified credential handle is no longer required by the application, and frees associated resources. Implementations are encouraged to set the cred_handle to GSS_C_NO_CREDENTIAL on successful completion of this call. Returns GSS_S_COMPLETE for successful completion, and GSS_S_NO_CRED for credentials could not be accessed.

3.3 Context-Level Routines

Table 2-2 GSS-API Context-Level Routines

Routine -----	Section -----	Function -----
<code>gss_init_sec_context</code>	5.19	Initiate a security context with a peer application
<code>gss_accept_sec_context</code>	5.1	Accept a security context initiated by a peer application
<code>gss_delete_sec_context</code>	5.9	Discard a security context
<code>gss_process_context_token</code>	5.25	Process a token on a security context from a peer application
<code>gss_context_time</code>	5.7	Determine for how long a context will remain valid
<code>gss_inquire_context</code>	5.20	Obtain information about a security context
<code>gss_wrap_size_limit</code>	5.34	Determine token-size limit for <code>gss_wrap</code> on a context
<code>gss_export_sec_context</code>	5.14	Transfer a security context to another process
<code>gss_import_sec_context</code>	5.17	Import a transferred context

`OM_uint32 gss_init_sec_context` (`OM_uint32 * minor_status`, `const` [Function]
`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: Mechanism specific status code.

initiator_cred_handle: Optional handle for credentials claimed. Supply `GSS_C_NO_CREDENTIAL` to act as a default initiator principal. If no default initiator is defined, the function will return `GSS_S_NO_CRED`.

context_handle: Context handle for new context. Supply `GSS_C_NO_CONTEXT` for first call; use value returned by first call in continuation calls. Resources associated with this context-handle must be released by the application after use with a call to `gss_delete_sec_context()`.

target_name: Name of target.

mech_type: Optional object ID of desired mechanism. Supply `GSS_C_NO_OID` to obtain an implementation specific default

req_flags: Contains various independent flags, each of which requests that the context support a specific service option. Symbolic names are provided for each flag, and the symbolic names corresponding to the required flags should be logically-ORed together to form the bit-mask value. See below for details.

time_req: Optional Desired number of seconds for which context should remain valid. Supply 0 to request a default validity period.

input_chan_bindings: Optional Application-specified bindings. Allows application to securely bind channel identification information to the security context. Specify GSS_C_NO_CHANNEL_BINDINGS if channel bindings are not used.

input_token: Optional (see text) Token received from peer application. Supply GSS_C_NO_BUFFER, or a pointer to a buffer containing the value GSS_C_EMPTY_BUFFER on initial call.

actual_mech_type: Optional actual mechanism used. The OID returned via this parameter will be a pointer to static storage that should be treated as read-only; In particular the application should not attempt to free it. Specify NULL if not required.

output_token: Token to be sent to peer application. If the length field of the returned buffer is zero, no token need be sent to the peer application. Storage associated with this buffer must be freed by the application after use with a call to `gss_release_buffer()`.

ret_flags: Optional various independent flags, each of which indicates that the context supports a specific service option. Specify NULL if not required. Symbolic names are provided for each flag, and the symbolic names corresponding to the required flags should be logically-ANDed with the *ret_flags* value to test whether a given option is supported by the context. See below for details.

time_rec: Optional number of seconds for which the context will remain valid. If the implementation does not support context expiration, the value GSS_C_INDEFINITE will be returned. Specify NULL if not required.

Initiates the establishment of a security context between the application and a remote peer. Initially, the *input_token* parameter should be specified either as GSS_C_NO_BUFFER, or as a pointer to a `gss.buffer_desc` object whose length field contains the value zero. The routine may return a *output_token* which should be transferred to the peer application, where the peer application will present it to `gss_accept_sec_context`. If no token need be sent, `gss_init_sec_context` will indicate this by setting the length field of the *output_token* argument to zero. To complete the context establishment, one or more reply tokens may be required from the peer application; if so, `gss_init_sec_context` will return a status containing the supplementary information bit GSS_S_CONTINUE_NEEDED. In this case, `gss_init_sec_context` should be called again when the reply token is received from the peer application, passing the reply token to `gss_init_sec_context` via the *input_token* parameters.

Portable applications should be constructed to use the token length and return status to determine whether a token needs to be sent or waited for. Thus a typical portable caller should always invoke

```
int context_established = 0;
gss_ctx_id_t context_hdl = GSS_C_NO_CONTEXT;
...
input_token->length = 0;

while (!context_established) {
```

```

maj_stat = gss_init_sec_context(&min_stat,
                                cred_hdl,
                                &context_hdl,
                                target_name,
                                desired_mech,
                                desired_services,
                                desired_time,
                                input_bindings,
                                input_token,
                                &actual_mech,
                                output_token,
                                &actual_services,
                                &actual_time);

if (GSS_ERROR(maj_stat)) {
    report_error(maj_stat, min_stat);
};

if (output_token->length != 0) {
    send_token_to_peer(output_token);
    gss_release_buffer(&min_stat, output_token)
};
if (GSS_ERROR(maj_stat)) {

    if (context_hdl != GSS_C_NO_CONTEXT)
        gss_delete_sec_context(&min_stat,
                                &context_hdl,
                                GSS_C_NO_BUFFER);

    break;
};

if (maj_stat & GSS_S_CONTINUE_NEEDED) {
    receive_token_from_peer(input_token);
} else {
    context_established = 1;
};
};

```

Whenever the routine returns a major status that includes the value `GSS_S_CONTINUE_NEEDED`, the context is not fully established and the following restrictions apply to the output parameters:

- The value returned via the `time_rec` parameter is undefined Unless the accompanying `ret_flags` parameter contains the bit `GSS_C_PROT_READY_FLAG`, indicating that per-message services may be applied in advance of a successful completion status, the value returned via the `actual_mech_type` parameter is undefined until the routine returns a major status value of `GSS_S_COMPLETE`.
- The values of the `GSS_C_DELEG_FLAG`, `GSS_C_MUTUAL_FLAG`, `GSS_C_REPLAY_FLAG`, `GSS_C_SEQUENCE_FLAG`, `GSS_C_CONF_FLAG`,

GSS_C_INTEG_FLAG and GSS_C_ANON_FLAG bits returned via the `ret_flags` parameter should contain the values that the implementation expects would be valid if context establishment were to succeed. In particular, if the application has requested a service such as delegation or anonymous authentication via the `req_flags` argument, and such a service is unavailable from the underlying mechanism, `gss_init_sec_context` should generate a token that will not provide the service, and indicate via the `ret_flags` argument that the service will not be supported. The application may choose to abort the context establishment by calling `gss_delete_sec_context` (if it cannot continue in the absence of the service), or it may choose to transmit the token and continue context establishment (if the service was merely desired but not mandatory).

- The values of the GSS_C_PROT_READY_FLAG and GSS_C_TRANS_FLAG bits within `ret_flags` should indicate the actual state at the time `gss_init_sec_context` returns, whether or not the context is fully established.
- GSS-API implementations that support per-message protection are encouraged to set the GSS_C_PROT_READY_FLAG in the final `ret_flags` returned to a caller (i.e. when accompanied by a GSS_S_COMPLETE status code). However, applications should not rely on this behavior as the flag was not defined in Version 1 of the GSS-API. Instead, applications should determine what per-message services are available after a successful context establishment according to the GSS_C_INTEG_FLAG and GSS_C_CONF_FLAG values.
- All other bits within the `ret_flags` argument should be set to zero.

If the initial call of `gss_init_sec_context()` fails, the implementation should not create a context object, and should leave the value of the `context_handle` parameter set to GSS_C_NO_CONTEXT to indicate this. In the event of a failure on a subsequent call, the implementation is permitted to delete the "half-built" security context (in which case it should set the `context_handle` parameter to GSS_C_NO_CONTEXT), but the preferred behavior is to leave the security context untouched for the application to delete (using `gss_delete_sec_context`).

During context establishment, the informational status bits GSS_S_OLD_TOKEN and GSS_S_DUPLICATE_TOKEN indicate fatal errors, and GSS-API mechanisms should always return them in association with a routine error of GSS_S_FAILURE. This requirement for pairing did not exist in version 1 of the GSS-API specification, so applications that wish to run over version 1 implementations must special-case these codes.

The `req_flags` values:

GSS_C_DELEG_FLAG

True - Delegate credentials to remote peer. False - Don't delegate.

GSS_C_MUTUAL_FLAG

True - Request that remote peer authenticate itself. False - Authenticate self to remote peer only.

GSS_C_REPLAY_FLAG

True - Enable replay detection for messages protected with `gss_wrap` or `gss_get_mic`. False - Don't attempt to detect replayed messages.

GSS_C_SEQUENCE_FLAG

True - Enable detection of out-of-sequence protected messages. False - Don't attempt to detect out-of-sequence messages.

GSS_C_CONF_FLAG

True - Request that confidentiality service be made available (via `gss_wrap`). False - No per-message confidentiality service is required.

GSS_C_INTEG_FLAG

True - Request that integrity service be made available (via `gss_wrap` or `gss_get_mic`). False - No per-message integrity service is required.

GSS_C_ANON_FLAG

True - Do not reveal the initiator's identity to the acceptor. False - Authenticate normally.

The `ret_flags` values:

GSS_C_DELEG_FLAG

True - Credentials were delegated to the remote peer. False - No credentials were delegated.

GSS_C_MUTUAL_FLAG

True - The remote peer has authenticated itself. False - Remote peer has not authenticated itself.

GSS_C_REPLAY_FLAG

True - replay of protected messages will be detected. False - replayed messages will not be detected.

GSS_C_SEQUENCE_FLAG

True - out-of-sequence protected messages will be detected. False - out-of-sequence messages will not be detected.

GSS_C_CONF_FLAG

True - Confidentiality service may be invoked by calling `gss_wrap` routine. False - No confidentiality service (via `gss_wrap`) available. `gss_wrap` will provide message encapsulation, data-origin authentication and integrity services only.

GSS_C_INTEG_FLAG

True - Integrity service may be invoked by calling either `gss_get_mic` or `gss_wrap` routines. False - Per-message integrity service unavailable.

GSS_C_ANON_FLAG

True - The initiator's identity has not been revealed, and will not be revealed if any emitted token is passed to the acceptor. False - The initiator's identity has been or will be authenticated normally.

GSS_C_PROT_READY_FLAG

True - Protection services (as specified by the states of the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG`) are available for use if the accompanying major status return value is either

GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED. False - Protection services (as specified by the states of the GSS_C_CONF_FLAG and GSS_C_INTEG_FLAG) are available only if the accompanying major status return value is GSS_S_COMPLETE.

GSS_C_TRANS_FLAG

True - The resultant security context may be transferred to other processes via a call to `gss_export_sec_context()`. False - The security context is not transferable.

All other bits should be set to zero.

Valid return values and their meaning:

GSS_S_COMPLETE: Successful completion.

GSS_S_CONTINUE_NEEDED: Indicates that a token from the peer application is required to complete the context, and that `gss_init_sec_context` must be called again with that token.

GSS_S_DEFECTIVE_TOKEN: Indicates that consistency checks performed on the `input_token` failed.

GSS_S_DEFECTIVE_CREDENTIAL: Indicates that consistency checks performed on the credential failed.

GSS_S_NO_CRED: The supplied credentials were not valid for context initiation, or the credential handle did not reference any credentials.

GSS_S_CREDENTIALS_EXPIRED: The referenced credentials have expired.

GSS_S_BAD_BINDINGS: The `input_token` contains different channel bindings to those specified via the `input_chan_bindings` parameter.

GSS_S_BAD_SIG: The `input_token` contains an invalid MIC, or a MIC that could not be verified.

GSS_S_OLD_TOKEN: The `input_token` was too old. This is a fatal error during context establishment.

GSS_S_DUPLICATE_TOKEN: The `input_token` is valid, but is a duplicate of a token already processed. This is a fatal error during context establishment.

GSS_S_NO_CONTEXT: Indicates that the supplied context handle did not refer to a valid context.

GSS_S_BAD_NAME_TYPE: The provided `target_name` parameter contained an invalid or unsupported type of name.

GSS_S_BAD_NAME: The provided `target_name` parameter was ill-formed.

GSS_S_BAD_MECH: The specified mechanism is not supported by the provided credential, or is unrecognized by the implementation.

```

OM_uint32 gss_accept_sec_context (OM_uint32 *minor_status,          [Function]
    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_rec, gss_cred_id_t
    *delegated_cred_handle)

```

minor_status: Integer, modify Mechanism specific status code.

context_handle: `gss_ctx_id_t`, read/modify context handle for new context. Supply `GSS_C_NO_CONTEXT` for first call; use value returned in subsequent calls. Once `gss_accept_sec_context()` has returned a value via this parameter, resources have been assigned to the corresponding context, and must be freed by the application after use with a call to `gss_delete_sec_context()`.

acceptor_cred_handle: `gss_cred_id_t`, read Credential handle claimed by context acceptor. Specify `GSS_C_NO_CREDENTIAL` to accept the context as a default principal. If `GSS_C_NO_CREDENTIAL` is specified, but no default acceptor principal is defined, `GSS_S_NO_CRED` will be returned.

input_token_buffer: buffer, opaque, read token obtained from remote application.

input_chan_bindings: channel bindings, read, optional Application- specified bindings. Allows application to securely bind channel identification information to the security context. If channel bindings are not used, specify `GSS_C_NO_CHANNEL_BINDINGS`.

src_name: `gss_name_t`, modify, optional Authenticated name of context initiator. After use, this name should be deallocated by passing it to `gss_release_name()`. If not required, specify `NULL`.

mech_type: Object ID, modify, optional Security mechanism used. The returned OID value will be a pointer into static storage, and should be treated as read-only by the caller (in particular, it does not need to be freed). If not required, specify `NULL`.

output_token: buffer, opaque, modify Token to be passed to peer application. If the length field of the returned token buffer is 0, then no token need be passed to the peer application. If a non- zero length field is returned, the associated storage must be freed after use by the application with a call to `gss_release_buffer()`.

ret_flags: bit-mask, modify, optional Contains various independent flags, each of which indicates that the context supports a specific service option. If not needed, specify `NULL`. Symbolic names are provided for each flag, and the symbolic names corresponding to the required flags should be logically-ANDed with the `ret_flags` value to test whether a given option is supported by the context. See below for the values.

time_rec: Integer, modify, optional number of seconds for which the context will remain valid. Specify `NULL` if not required.

delegated_cred_handle: `gss_cred_id_t`, modify, optional credential handle for credentials received from context initiator. Only valid if `deleg_flag` in `ret_flags` is true, in which case an explicit credential handle (i.e. not `GSS_C_NO_CREDENTIAL`) will be returned; if `deleg_flag` is false, `gss_accept_context()` will set this parameter to `GSS_C_NO_CREDENTIAL`. If a credential handle is returned, the associated re-

sources must be released by the application after use with a call to `gss_release_cred()`. Specify `NULL` if not required.

Allows a remotely initiated security context between the application and a remote peer to be established. The routine may return a `output_token` which should be transferred to the peer application, where the peer application will present it to `gss_init_sec_context`. If no token need be sent, `gss_accept_sec_context` will indicate this by setting the length field of the `output_token` argument to zero. To complete the context establishment, one or more reply tokens may be required from the peer application; if so, `gss_accept_sec_context` will return a status flag of `GSS_S_CONTINUE_NEEDED`, in which case it should be called again when the reply token is received from the peer application, passing the token to `gss_accept_sec_context` via the `input_token` parameters.

Portable applications should be constructed to use the token length and return status to determine whether a token needs to be sent or waited for. Thus a typical portable caller should always invoke `gss_accept_sec_context` within a loop:

```

gss_ctx_id_t context_hdl = GSS_C_NO_CONTEXT;

do {
    receive_token_from_peer(input_token);
    maj_stat = gss_accept_sec_context(&min_stat,
                                     &context_hdl,
                                     cred_hdl,
                                     input_token,
                                     input_bindings,
                                     &client_name,
                                     &mech_type,
                                     output_token,
                                     &ret_flags,
                                     &time_rec,
                                     &deleg_cred);

    if (GSS_ERROR(maj_stat)) {
        report_error(maj_stat, min_stat);
    };
    if (output_token->length != 0) {
        send_token_to_peer(output_token);

        gss_release_buffer(&min_stat, output_token);
    };
    if (GSS_ERROR(maj_stat)) {
        if (context_hdl != GSS_C_NO_CONTEXT)
            gss_delete_sec_context(&min_stat,
                                   &context_hdl,
                                   GSS_C_NO_BUFFER);

        break;
    };
} while (maj_stat & GSS_S_CONTINUE_NEEDED);

```

Whenever the routine returns a major status that includes the value `GSS_S_CONTINUE_NEEDED`, the context is not fully established and the following restrictions apply to the output parameters:

The value returned via the `time_rec` parameter is undefined Unless the accompanying `ret_flags` parameter contains the bit `GSS_C_PROT_READY_FLAG`, indicating that per-message services may be applied in advance of a successful completion status, the value returned via the `mech_type` parameter may be undefined until the routine returns a major status value of `GSS_S_COMPLETE`.

The values of the `GSS_C_DELEG_FLAG`, `GSS_C_MUTUAL_FLAG`, `GSS_C_REPLAY_FLAG`, `GSS_C_SEQUENCE_FLAG`, `GSS_C_CONF_FLAG`, `GSS_C_INTEG_FLAG` and `GSS_C_ANON_FLAG` bits returned via the `ret_flags` parameter should contain the values that the implementation expects would be valid if context establishment were to succeed.

The values of the `GSS_C_PROT_READY_FLAG` and `GSS_C_TRANS_FLAG` bits within `ret_flags` should indicate the actual state at the time `gss_accept_sec_context` returns, whether or not the context is fully established.

Although this requires that GSS-API implementations set the `GSS_C_PROT_READY_FLAG` in the final `ret_flags` returned to a caller (i.e. when accompanied by a `GSS_S_COMPLETE` status code), applications should not rely on this behavior as the flag was not defined in Version 1 of the GSS-API. Instead, applications should be prepared to use per-message services after a successful context establishment, according to the `GSS_C_INTEG_FLAG` and `GSS_C_CONF_FLAG` values.

All other bits within the `ret_flags` argument should be set to zero. While the routine returns `GSS_S_CONTINUE_NEEDED`, the values returned via the `ret_flags` argument indicate the services that the implementation expects to be available from the established context.

If the initial call of `gss_accept_sec_context()` fails, the implementation should not create a context object, and should leave the value of the `context_handle` parameter set to `GSS_C_NO_CONTEXT` to indicate this. In the event of a failure on a subsequent call, the implementation is permitted to delete the "half-built" security context (in which case it should set the `context_handle` parameter to `GSS_C_NO_CONTEXT`), but the preferred behavior is to leave the security context (and the `context_handle` parameter) untouched for the application to delete (using `gss_delete_sec_context`).

During context establishment, the informational status bits `GSS_S_OLD_TOKEN` and `GSS_S_DUPLICATE_TOKEN` indicate fatal errors, and GSS-API mechanisms should always return them in association with a routine error of `GSS_S_FAILURE`. This requirement for pairing did not exist in version 1 of the GSS-API specification, so applications that wish to run over version 1 implementations must special-case these codes.

The `ret_flags` flag values:

`GSS_C_DELEG_FLAG`

True - Delegated credentials are available via the `delegated_cred_handle` parameter. False - No credentials were delegated.

GSS_C_MUTUAL_FLAG

True - Remote peer asked for mutual authentication. False - Remote peer did not ask for mutual authentication.

GSS_C_REPLAY_FLAG

True - replay of protected messages will be detected. False - replayed messages will not be detected.

GSS_C_SEQUENCE_FLAG

True - out-of-sequence protected messages will be detected. False - out-of-sequence messages will not be detected.

GSS_C_CONF_FLAG

True - Confidentiality service may be invoked by calling the `gss_wrap` routine. False - No confidentiality service (via `gss_wrap`) available. `gss_wrap` will provide message encapsulation, data-origin authentication and integrity services only.

GSS_C_INTEG_FLAG

True - Integrity service may be invoked by calling either `gss_get_mic` or `gss_wrap` routines. False - Per-message integrity service unavailable.

GSS_C_ANON_FLAG

True - The initiator does not wish to be authenticated; the `src_name` parameter (if requested) contains an anonymous internal name. False - The initiator has been authenticated normally.

GSS_C_PROT_READY_FLAG

True - Protection services (as specified by the states of the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG`) are available if the accompanying major status return value is either `GSS_S_COMPLETE` or `GSS_S_CONTINUE_NEEDED`. False - Protection services (as specified by the states of the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG`) are available only if the accompanying major status return value is `GSS_S_COMPLETE`.

GSS_C_TRANS_FLAG

True - The resultant security context may be transferred to other processes via a call to `gss_export_sec_context()`. False - The security context is not transferable.

All other bits should be set to zero.

Return values:

GSS_S_CONTINUE_NEEDED: Indicates that a token from the peer application is required to complete the context, and that `gss_accept_sec_context` must be called again with that token.

GSS_S_DEFECTIVE_TOKEN: Indicates that consistency checks performed on the `input_token` failed.

GSS_S_DEFECTIVE_CREDENTIAL: Indicates that consistency checks performed on the credential failed.

GSS_S_NO_CRED: The supplied credentials were not valid for context acceptance, or the credential handle did not reference any credentials.

GSS_S_CREDENTIALS_EXPIRED: The referenced credentials have expired.

GSS_S_BAD_BINDINGS: The `input_token` contains different channel bindings to those specified via the `input_chan_bindings` parameter.

GSS_S_NO_CONTEXT: Indicates that the supplied context handle did not refer to a valid context.

GSS_S_BAD_SIG: The `input_token` contains an invalid MIC.

GSS_S_OLD_TOKEN: The `input_token` was too old. This is a fatal error during context establishment.

GSS_S_DUPLICATE_TOKEN: The `input_token` is valid, but is a duplicate of a token already processed. This is a fatal error during context establishment.

GSS_S_BAD_MECH: The received token specified a mechanism that is not supported by the implementation or the provided credential.

OM_uint32 gss_delete_sec_context (OM_uint32 * *minor_status*, [Function]
 gss_ctx_id_t * *context_handle*, gss_buffer_t *output_token*)

minor_status: Mechanism specific status code.

context_handle: Context handle identifying context to delete. After deleting the context, the GSS-API will set this context handle to GSS_C_NO_CONTEXT.

output_token: Optional token to be sent to remote application to instruct it to also delete the context. It is recommended that applications specify GSS_C_NO_BUFFER for this parameter, requesting local deletion only. If a buffer parameter is provided by the application, the mechanism may return a token in it; mechanisms that implement only local deletion should set the length field of this token to zero to indicate to the application that no token is to be sent to the peer.

Delete a security context. `gss_delete_sec_context()` will delete the local data structures associated with the specified security context, and may generate an `output_token`, which when passed to the peer `gss_process_context_token()` will instruct it to do likewise. If no token is required by the mechanism, the GSS-API should set the length field of the `output_token` (if provided) to zero. No further security services may be obtained using the context specified by `context_handle`.

In addition to deleting established security contexts, `gss_delete_sec_context()` must also be able to delete "half-built" security contexts resulting from an incomplete sequence of `gss_init_sec_context()/gss_accept_sec_context()` calls.

The `output_token` parameter is retained for compatibility with version 1 of the GSS-API. It is recommended that both peer applications invoke `gss_delete_sec_context()` passing the value GSS_C_NO_BUFFER for the `output_token` parameter, indicating that no token is required, and that `gss_delete_sec_context()` should simply delete local context data structures. If the application does pass a valid buffer to `gss_delete_sec_context()`, mechanisms are encouraged to return a zero-length token, indicating that no peer action is necessary, and that no token should be transferred by the application.

Returns GSS_S_COMPLETE for successful completion, and GSS_S_NO_CONTEXT if no valid context was supplied.

3.4 Per-Message Routines

Table 2-3 GSS-API Per-message Routines

Routine -----	Section -----	Function -----
<code>gss_get_mic</code>	5.15	Calculate a cryptographic message integrity code (MIC) for a message; integrity service
<code>gss_verify_mic</code>	5.32	Check a MIC against a message; verify integrity of a received message
<code>gss_wrap</code>	5.33	Attach a MIC to a message, and optionally encrypt the message content; confidentiality service
<code>gss_unwrap</code>	5.31	Verify a message with attached MIC, and decrypt message content if necessary.

`OM_uint32 gss_wrap` (`OM_uint32 * minor_status`, const [Function]
`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: Mechanism specific status code.

context_handle: Identifies the context on which the message will be sent

conf_req_flag: Whether confidentiality is requested.

qop_req: Specifies required quality of protection. A mechanism-specific default may be requested by setting `qop_req` to `GSS_C_QOP_DEFAULT`. If an unsupported protection strength is requested, `gss_wrap` will return a `major_status` of `GSS_S_BAD_QOP`.

input_message_buffer: Message to be protected.

conf_state: Optional output variable indicating if confidentiality services have been applied.

output_message_buffer: Buffer to receive protected message. Storage associated with this message must be freed by the application after use with a call to `gss_release_buffer()`.

Attaches a cryptographic MIC and optionally encrypts the specified `input_message`. The `output_message` contains both the MIC and the message. The `qop_req` parameter allows a choice between several cryptographic algorithms, if supported by the chosen mechanism.

Since some application-level protocols may wish to use tokens emitted by `gss_wrap()` to provide "secure framing", implementations must support the wrapping of zero-length messages.

Returns

GSS_S_COMPLETE Successful completion

GSS_S_CONTEXT_EXPIRED The context has already expired

GSS_S_NO_CONTEXT The context_handle parameter did not identify a valid context

GSS_S_BAD_QOP The specified QOP is not supported by the mechanism.

OM_uint32 gss_unwrap (OM_uint32 * *minor_status*, const [Function]
 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: Mechanism specific status code.

context_handle: Identifies the context on which the message arrived

input_message_buffer: input protected message

output_message_buffer: Buffer to receive unwrapped message. Storage associated with this buffer must be freed by the application after use with a call to `gss_release_buffer()`.

conf_state: optional output variable indicating if confidentiality protection was used.

qop_state: optional output variable indicating quality of protection.

Converts a message previously protected by `gss_wrap` back to a usable form, verifying the embedded MIC. The *conf_state* parameter indicates whether the message was encrypted; the *qop_state* parameter indicates the strength of protection that was used to provide the confidentiality and integrity services.

Since some application-level protocols may wish to use tokens emitted by `gss_wrap()` to provide "secure framing", implementations must support the wrapping and unwrapping of zero-length messages.

Returns:

GSS_S_COMPLETE Successful completion

GSS_S_DEFECTIVE_TOKEN The token failed consistency checks

GSS_S_BAD_SIG The MIC was incorrect

GSS_S_DUPLICATE_TOKEN The token was valid, and contained a correct MIC for the message, but it had already been processed

GSS_S_OLD_TOKEN The token was valid, and contained a correct MIC for the message, but it is too old to check for duplication.

GSS_S_UNSEQ_TOKEN The token was valid, and contained a correct MIC for the message, but has been verified out of sequence; a later token has already been received.

GSS_S_GAP_TOKEN The token was valid, and contained a correct MIC for the message, but has been verified out of sequence; an earlier expected token has not yet been received.

GSS_S_CONTEXT_EXPIRED The context has already expired

GSS_S_NO_CONTEXT The context_handle parameter did not identify a valid context

3.5 Name Manipulation

Table 2-4 GSS-API Name manipulation Routines

Routine	Section	Function
-----	-----	-----
<code>gss_import_name</code>	5.16	Convert a contiguous string name to internal-form
<code>gss_display_name</code>	5.10	Convert internal-form name to text
<code>gss_compare_name</code>	5.6	Compare two internal-form names
<code>gss_release_name</code>	5.28	Discard an internal-form name
<code>gss_inquire_names_for_mech</code>	5.24	List the name-types supported by the specified mechanism
<code>gss_inquire_mechs_for_name</code>	5.23	List mechanisms that support the specified name-type
<code>gss_canonicalize_name</code>	5.5	Convert an internal name to an MN
<code>gss_export_name</code>	5.13	Convert an MN to export form
<code>gss_duplicate_name</code>	5.12	Create a copy of an internal name

`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`) [Function]

minor_status: Mechanism specific status code

input_name_buffer: buffer containing contiguous string name to convert

input_name_type: Optional Object ID specifying type of printable name. Applications may specify either `GSS_C_NO_OID` to use a mechanism-specific default printable syntax, or an OID recognized by the GSS-API implementation to name a specific namespace.

output_name: returned name in internal form. Storage associated with this name must be freed by the application after use with a call to `gss_release_name()`.

Convert a contiguous string name to internal form. In general, the internal name returned (via the `<output_name>` parameter) will not be an MN; the exception to this is if the `<input_name_type>` indicates that the contiguous string provided via the `<input_name_buffer>` parameter is of type `GSS_C_NT_EXPORT_NAME`, in which case the returned internal name will be an MN for the mechanism that exported the name.

Returns `GSS_S_COMPLETE` for successful completion, `GSS_S_BAD_NAME_TYPE` when the `input_name_type` was unrecognized, `GSS_S_BAD_NAME` when the `input_name` parameter could not be interpreted as a name of the specified type, and `GSS_S_BAD_MECH` when the `input_name_type` was `GSS_C_NT_EXPORT_NAME`, but the mechanism contained within the `input_name` is not supported.

OM_uint32 gss_display_name (OM_uint32 * *minor_status*, const [Function]
 gss_name_t *input_name*, gss_buffer_t *output_name_buffer*, gss_OID *
output_name_type)

minor_status: Mechanism specific status code.

input_name: Name to be displayed

output_name_buffer: Buffer to receive textual name string. The application must free storage associated with this name after use with a call to `gss_release_buffer()`.

output_name_type: Optional type of the returned name. The returned gss_OID will be a pointer into static storage, and should be treated as read-only by the caller (in particular, the application should not attempt to free it). Specify NULL if not required.

Allows an application to obtain a textual representation of an opaque internal-form name for display purposes. The syntax of a printable name is defined by the GSS-API implementation.

If *input_name* denotes an anonymous principal, the implementation should return the gss_OID value GSS_C_NT_ANONYMOUS as the *output_name_type*, and a textual name that is syntactically distinct from all valid supported printable names in *output_name_buffer*.

If *input_name* was created by a call to `gss_import_name`, specifying GSS_C_NO_OID as the name-type, implementations that employ lazy conversion between name types may return GSS_C_NO_OID via the *output_name_type* parameter.

Returns GSS_S_COMPLETE for successful completion, GSS_S_BAD_NAME when *input_name* was ill-formed.

OM_uint32 gss_compare_name (OM_uint32 * *minor_status*, const [Function]
 gss_name_t *name1*, const gss_name_t *name2*, int * *name_equal*)

minor_status: Mechanism specific status code.

name1: Internal-form name.

name2: Internal-form name.

name_equal: non-zero if names refer to same entity.

Allows an application to compare two internal-form names to determine whether they refer to the same entity.

If either name presented to `gss_compare_name` denotes an anonymous principal, the routines should indicate that the two names do not refer to the same identity.

Returns GSS_S_COMPLETE for successful completion, GSS_S_BAD_NAME_TYPE when the two names were of incomparable types, and GSS_S_BAD_NAME if one or both of *name1* or *name2* was ill-formed.

OM_uint32 gss_release_name (OM_uint32 * *minor_status*, [Function]
 gss_name_t * *name*)

minor_status: Mechanism specific status code.

name: The name to be deleted.

Free GSSAPI-allocated storage associated with an internal-form name. Implementations are encouraged to set the name to GSS_C_NO_NAME on successful completion of this call.

Returns GSS_S_COMPLETE for successful completion, and GSS_S_BAD_NAME when the name parameter did not contain a valid name.

OM_uint32 gss_canonicalize_name (OM_uint32 * *minor_status*, [Function]
 const gss_name_t *input_name*, const gss_OID *mech_type*, gss_name_t *
output_name)

minor_status: Mechanism specific status code.

input_name: The name for which a canonical form is desired.

mech_type: The authentication mechanism for which the canonical form of the name is desired. The desired mechanism must be specified explicitly; no default is provided.

output_name: The resultant canonical name. Storage associated with this name must be freed by the application after use with a call to `gss_release_name()`.

Generate a canonical mechanism name (MN) from an arbitrary internal name. The mechanism name is the name that would be returned to a context acceptor on successful authentication of a context where the initiator used the *input_name* in a successful call to `gss_acquire_cred`, specifying an OID set containing <*mech_type*> as its only member, followed by a call to `gss_init_sec_context`, specifying <*mech_type*> as the authentication mechanism.

Returns

GSS_S_COMPLETE Successful completion.

GSS_S_BAD_MECH The identified mechanism is not supported.

GSS_S_BAD_NAME_TYPE The provided internal name contains no elements that could be processed by the specified mechanism.

GSS_S_BAD_NAME The provided internal name was ill-formed.

OM_uint32 gss_inquire_names_for_mech (OM_uint32 [Function]
 **minor_status*, const gss_OID *mechanism*, gss_OID_set **name_types*)

minor_status: Implementation specific status code.

mechanism: The mechanism to be interrogated.

name_types: Output set of name-types supported by the specified mechanism. The returned OID set must be freed by the application after use with a call to `gss_release_oid_set()`.

Outputs the set of nametypes supported by the specified mechanism.

Returns GSS_S_COMPLETE for successful completion.

OM_uint32 gss_inquire_mechs_for_name (OM_uint32 [Function]
 **minor_status*, const gss_name_t *input_name*, gss_OID_set **mech_types*)

minor_status: Implementation specific status code.

input_name: The name to which the inquiry relates.

mech_types: Output set of mechanisms that may support the specified name. The returned OID set must be freed by the caller after use with a call to `gss_release_oid_set()`.

Outputs the set of mechanisms supported by the GSS-API implementation that may be able to process the specified name.

Each mechanism returned will recognize at least one element within the name. It is permissible for this routine to be implemented within a mechanism-independent GSS-API layer, using the type information contained within the presented name, and based on registration information provided by individual mechanism implementations. This means that the returned `mech_types` set may indicate that a particular mechanism will understand the name when in fact it would refuse to accept the name as input to `gss_canonicalize_name()`, `gss_init_sec_context()`, `gss_acquire_cred()` or `gss_add_cred()` (due to some property of the specific name, as opposed to the name type). Thus this routine should be used only as a pre-filter for a call to a subsequent mechanism-specific routine.

Returns `GSS_S_COMPLETE` for successful completion, `GSS_S_BAD_NAME` to indicate that the `input_name` parameter was ill-formed, and `GSS_S_BAD_NAME_TYPE` to indicate that the `input_name` parameter contained an invalid or unsupported type of name.

```
OM_uint32 gss_canonicalize_name (OM_uint32 *minor_status,          [Function]
                                const gss_name_t input_name, const gss_OID mech_type, gss_name_t
                                *output_name)
```

minor_status: Mechanism specific status code.

input_name: The name for which a canonical form is desired.

mech_type: The authentication mechanism for which the canonical form of the name is desired. The desired mechanism must be specified explicitly; no default is provided.

output_name: The resultant canonical name. Storage associated with this name must be freed by the application after use with a call to `gss_release_name()`.

Generate a canonical mechanism name (MN) from an arbitrary internal name. The mechanism name is the name that would be returned to a context acceptor on successful authentication of a context where the initiator used the `input_name` in a successful call to `gss_acquire_cred`, specifying an OID set containing `<mech_type>` as its only member, followed by a call to `gss_init_sec_context`, specifying `<mech_type>` as the authentication mechanism.

Returns `GSS_S_COMPLETE` for successful completion, `GSS_S_BAD_MECH` to indicate that the identified mechanism is not supported, `GSS_S_BAD_NAME_TYPE` to indicate that the provided internal name contains no elements that could be processed by the specified mechanism, and `GSS_S_BAD_NAME` to indicate that the provided internal name was ill-formed.

```
OM_uint32 gss_export_name (OM_uint32 *minor_status, const          [Function]
                           gss_name_t input_name, gss_buffer_t exported_name)
```

minor_status: Mechanism specific status code.

input_name: The mechanism name to be exported.

exported_name: Output variable with canonical contiguous string form of `input_name`. Storage associated with this string must be freed by the application after use with `gss_release_buffer()`.

To produce a canonical contiguous string representation of a mechanism name (MN), suitable for direct comparison (e.g. with `memcmp`) for use in authorization functions

(e.g. matching entries in an access-control list). The *input_name* parameter must specify a valid MN (i.e. an internal name generated by `gss_accept_sec_context` or by `gss_canonicalize_name`).

Returns `GSS_S_COMPLETE` for successful completion, `GSS_S_NAME_NOT_MN` to indicate that the provided internal name was not a mechanism name, `GSS_S_BAD_NAME` to indicate that the provided internal name was ill-formed, and `GSS_S_BAD_NAME_TYPE` to indicate that the internal name was of a type not supported by the GSS-API implementation.

`OM_uint32 gss_duplicate_name (OM_uint32 * minor_status, const [Function]
gss_name_t src_name, gss_name_t * dest_name)`

minor_status: Mechanism specific status code.

src_name: Internal name to be duplicated.

dest_name: The resultant copy of `<src_name>`. Storage associated with this name must be freed by the application after use with a call to `gss_release_name()`.

Create an exact duplicate of the existing internal name *src_name*. The new *dest_name* will be independent of *src_name* (i.e. *src_name* and *dest_name* must both be released, and the release of one shall not affect the validity of the other).

Returns `GSS_S_COMPLETE` for successful completion, and `GSS_S_BAD_NAME` when the *src_name* parameter was ill-formed.

3.6 Miscellaneous Routines

Table 2-5 GSS-API Miscellaneous Routines

Routine -----	Section -----	Function -----
<code>gss_add_oid_set_member</code>	5.4	Add an object identifier to a set
<code>gss_display_status</code>	5.11	Convert a GSS-API status code to text
<code>gss_indicate_mechs</code>	5.18	Determine available underlying authentication mechanisms
<code>gss_release_buffer</code>	5.26	Discard a buffer
<code>gss_release_oid_set</code>	5.29	Discard a set of object identifiers
<code>gss_create_empty_oid_set</code>	5.8	Create a set containing no object identifiers
<code>gss_test_oid_set_member</code>	5.30	Determines whether an object identifier is a member of a set.

`OM_uint32 gss_release_buffer (OM_uint32 * minor_status, [Function]
gss_buffer_t buffer)`

minor_status: Mechanism specific status code.

buffer: The storage associated with the buffer will be deleted. The `gss_buffer_desc` object will not be freed, but its `length` field will be zeroed.

Free storage associated with a buffer. The storage must have been allocated by a GSS-API routine. In addition to freeing the associated storage, the routine will zero the length field in the descriptor to which the buffer parameter refers, and implementations are encouraged to additionally set the pointer field in the descriptor to NULL. Any buffer object returned by a GSS-API routine may be passed to `gss_release_buffer` (even if there is no storage associated with the buffer).

Returns `GSS_S_COMPLETE` for successful completion.

`OM_uint32 gss_create_empty_oid_set` (`OM_uint32 * minor_status,` [Function]
`gss_OID_set * oid_set`)

minor_status: Mechanism specific status code

oid_set: The empty object identifier set. The routine will allocate the `gss_OID_set_desc` object, which the application must free after use with a call to `gss_release_oid_set()`.

Create an object-identifier set containing no object identifiers, to which members may be subsequently added using the `gss_add_oid_set_member()` routine. These routines are intended to be used to construct sets of mechanism object identifiers, for input to `gss_acquire_cred`.

Returns `GSS_S_COMPLETE` for successful completion.

`OM_uint32 gss_add_oid_set_member` (`OM_uint32 * minor_status,` [Function]
`const gss_OID member_oid, gss_OID_set * oid_set`)

minor_status: Mechanism specific status code

member_oid: The object identifier to copied into the set.

oid_set: The set in which the object identifier should be inserted.

Add an Object Identifier to an Object Identifier set. This routine is intended for use in conjunction with `gss_create_empty_oid_set` when constructing a set of mechanism OIDs for input to `gss_acquire_cred`. The `oid_set` parameter must refer to an OID-set that was created by GSS-API (e.g. a set returned by `gss_create_empty_oid_set()`). GSS-API creates a copy of the `member_oid` and inserts this copy into the set, expanding the storage allocated to the OID-set's elements array if necessary. The routine may add the new member OID anywhere within the elements array, and implementations should verify that the new `member_oid` is not already contained within the elements array; if the `member_oid` is already present, the `oid_set` should remain unchanged.

Returns `GSS_S_COMPLETE` for successful completion.

`OM_uint32 gss_test_oid_set_member` (`OM_uint32 * minor_status,` [Function]
`const gss_OID member, const gss_OID_set set, int * present`)

minor_status: Mechanism specific status code

member: The object identifier whose presence is to be tested.

set: The Object Identifier set.

present: output indicating if the specified OID is a member of the set, zero if not.

Interrogate an Object Identifier set to determine whether a specified Object Identifier is a member. This routine is intended to be used with OID sets returned by `gss_indicate_mechs()`, `gss_acquire_cred()`, and `gss_inquire_cred()`, but will also work with user-generated sets.

Returns `GSS_S_COMPLETE` for successful completion.

`OM_uint32 gss_release_oid_set` (`OM_uint32 *minor_status,` [Function]
`gss_OID_set *set`)

minor_status: Mechanism specific status code

set: The storage associated with the `gss_OID_set` will be deleted.

Free storage associated with a GSSAPI-generated `gss_OID_set` object. The set parameter must refer to an OID-set that was returned from a GSS-API routine. `gss_release_oid_set()` will free the storage associated with each individual member OID, the OID set's elements array, and the `gss_OID_set_desc`.

Implementations are encouraged to set the `gss_OID_set` parameter to `GSS_C_NO_OID_SET` on successful completion of this routine.

Returns `GSS_S_COMPLETE` for successful completion.

`OM_uint32 gss_indicate_mechs` (`OM_uint32 *minor_status,` [Function]
`gss_OID_set *mech_set`)

minor_status: Mechanism specific status code.

mech_set: Output OID set with implementation-supported mechanisms.

Allows an application to determine which underlying security mechanisms are available.

The returned `gss_OID_set` value will be a dynamically-allocated OID set, that should be released by the caller after use with a call to `gss_release_oid_set()`.

Returns `GSS_S_COMPLETE` for successful completion.

`OM_uint32 gss_display_status` (`OM_uint32 *minor_status,` [Function]
`OM_uint32 status_value, int status_type, const gss_OID mech_type,`
`OM_uint32 *message_context, gss_buffer_t status_string`)

minor_status: Mechanism specific status code.

status_value Status value to be converted

status_type: Type of status code. Valid values include `GSS_C_GSS_CODE` to indicate that `status_value` is a GSS status code, and `GSS_C_MECH_CODE` to indicate that `status_value` is a mechanism status code.

mech_type: Optional OID of underlying mechanism (used to interpret a minor status value) Supply `GSS_C_NO_OID` to obtain the system default.

message_context: Input/output variable that should be initialized to zero by the application prior to the first call. On return from `gss_display_status()`, a non-zero `status_value` parameter indicates that additional messages may be extracted from the status code via subsequent calls to `gss_display_status()`, passing the same `status_value`, `status_type`, `mech_type`, and `message_context` parameters.

status_string: Output textual interpretation of the *status_value*. Storage associated with this parameter must be freed by the application after use with a call to `gss_release_buffer()`.

Allows an application to obtain a textual representation of a GSS-API status code, for display to the user or for logging purposes. Since some status values may indicate multiple conditions, applications may need to call `gss_display_status` multiple times, each call generating a single text string. The *message_context* parameter is used by `gss_display_status` to store state information about which error messages have already been extracted from a given *status_value*; *message_context* must be initialized to 0 by the application prior to the first call, and `gss_display_status` will return a non-zero value in this parameter if there are further messages to extract.

The *message_context* parameter contains all state information required by `gss_display_status` in order to extract further messages from the *status_value*; even when a non-zero value is returned in this parameter, the application is not required to call `gss_display_status` again unless subsequent messages are desired. The following code extracts all messages from a given status code and prints them to `stderr`:

```

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);
```

Returns `GSS_S_COMPLETE` for successful completion, `GSS_S_BAD_MECH` to indicate that translation in accordance with an unsupported mechanism type was requested, and `GSS_S_BAD_STATUS` to indicate that the status value was not recognized, or the status type was neither `GSS_C_GSS_CODE` nor `GSS_C_MECH_CODE`.

4 Extended GSS API

None of the following functions are standard GSS API functions. As such, they are not declared in `'gss/api.h'`, but rather in `'gss/ext.h'` (which is included from `'gss.h'`).

const char * gss_check_version (const char * *req_version*) [Function]
req_version: version string to compare with, or NULL

Check that the the version of the library is at minimum the one given as a string in *req_version* and return the actual version string of the library; return NULL if the condition is not met. If *NULL* is passed to this function no check is done and only the version string is returned. It is a pretty good idea to run this function as soon as possible, because it may also intializes some subsystems. In a multithreaded environment if should be called before any more threads are created.

int gss_oid_equal (gss_OID *first_oid*, gss_OID *second_oid*) [Function]
 Compare two OIDs for equality. Compares actual content, not just pointer equality. Returns a boolean true iff the OIDs are equal.

OM_uint32 gss_copy_oid (OM_uint32 * *minor_status*, const gss_OID [Function]
src_oid, gss_OID *dest_oid*);

Make an exact copy of the given OID, that shares no memory areas with the original. The contents of the copied OID must be deallocated by the caller. Returns GSS_S_COMPLETE on success.

OM_uint32 gss_duplicate_oid (OM_uint32 * *minor_status*, const [Function]
 gss_OID *src_oid*, gss_OID * *dest_oid*)

Allocate an exact copy of the given OID, that shares no memory areas with the original. The newly created OID, and its contents, must be deallocated by the caller. Returns GSS_S_COMPLETE on success.

int gss_encapsulate_token (gss_buffer_t *input_message*, gss_OID [Function]
token_oid, gss_buffer_t *output_message*)

input_message: Message to be encapsulated.

token_oid: OID of mechanism.

input_message: Output buffer with encapsulated message.

Wrap a buffer in the mechanism-independent token format. This is used for the initial token of a GSS-API context establishment sequence. It incorporates an identifier of the mechanism type to be used on that context, and enables tokens to be interpreted unambiguously at GSS-API peers. See further section 3.1 of RFC 2743.

int gss_decapsulate_token (gss_buffer_t *input_message*, gss_OID [Function]
token_oid, gss_buffer_t *output_message*)

input_message: Message to decapsulated.

token_oid: Output buffer with mechanism OID used in message.

input_message: Output buffer with encapsulated message.

Unwrap a buffer in the mechanism-independent token format. This is the reverse of `gss_encapsulate_token`. The translation is loss-less, all data is preserved as is.

5 Acknowledgements

This manual borrows text from RFC 2743 and RFC 2744 that describe GSS API formally.

Appendix A Criticism of GSS

The author has doubts whether GSS is the best solution for free software projects looking for a implementation agnostic security framework. We express these doubts in this section, so that the reader can judge for herself if any of the potential problems discussed here are relevant for their project, or if the benefit outweigh the problems. GSS can be criticized on several levels. We start with the actual implementation.

GSS does not appear to be designed by experienced C programmers. While generally this may be a good thing (C is not the best language), but since they defined the API in C, it is unfortunate. The primary evidence of this is the `major_status` and `minor_status` error code solution. It is a complicated way to describe error conditions, but what makes matters worse, the error condition is separated; half of the error condition is in the function return value and the other half is in the first argument to the function, which is always a pointer to an integer. (The pointer is not even allowed to be `NULL`, if the application doesn't care about the minor error code.) This makes the API unreadable, and difficult to use. A better solutions would be to return a struct containing the entire error condition, which can be accessed using macros, although we acknowledge that the C language used at the time GSS was designed may not have allowed this (this may in fact be the reason the awkward solution was chosen). Instead, the return value could have been passed back to callers using a pointer to a struct, accessible using various macros, and the function could have a void prototype. The fact that `minor_status` is placed first in the parameter list increases the pain it is to use the API. Important parameters should be placed first. A better place for `minor_status` (if it must be present at all) would have been last in the prototypes.

Another evidence of the C inexperience are the memory management issues; GSS provides functions to deallocate data stored within, e.g., `gss_buffer_t` but the caller is responsible of deallocating the structure pointed at by the `gss_buffer_t` (i.e., the `gss_buffer_desc`) itself. Memory management issues are error prone, and this division easily leads to memory leaks (or worse). Instead, the API should be the sole owner of all `gss_ctx_id_t`, `gss_cred_id_t`, and `gss_buffer_t` structures: they should be allocated by the library, and deallocated (using the utility functions defined for this purpose) by the library.

TBA: thread issues

TBA: multiple mechanisms in a GSS library

TBA: high-level design criticism.

TBA: no credential forwarding.

TBA: internationalization

TBA: krb5: no way to access authorization-data

TBA: krb5: firewall/pre-IP: iakerb status?

TBA: krb5: single-DES only

TBA: the API may block, unusable in `select()` based servers. Especially if the servers contacted is decided by the, yet unauthenticated, remote client.

Finally we note that few free security applications uses GSS, perhaps the only major exception to this are Kerberos 5 implementations. While not substantial evidence, this do suggest that the GSS may not be the simplest solution available to solve actual problems,

since otherwise more projects would have chosen to take advantage of the work that went into GSS instead of using another framework (or designing their own solution).

Our conclusion is that free software projects that are looking for a security framework should evaluate carefully whether GSS actually is the best solution before using it. In particular it is recommended to compare GSS with the Simple Authentication and Security Layer (SASL) framework, which in several situations provide the same feature as GSS does. The most compelling argument for SASL over GSS is, as its acronym suggest, Simple, whereas GSS is far from it.

Concept Index

A

Aborting execution 6
AIX 2

C

Compiling your application 5

D

Debian 2

F

FreeBSD 3

H

Header files 4
HP-UX 2

I

IRIX 2

M

Mandrake 2
mechanism status codes 7
Memory allocation failure 6

N

NetBSD 3

O

OpenBSD 3
Out of Memory handling 6

R

RedHat 2
RedHat Advanced Server 2
Reporting Bugs 3

S

Solaris 2
status codes 7
SuSE 2
SuSE Linux 2

T

Tru64 2

W

Windows 2

API Index

G

<code>gss_accept_sec_context</code>	21	<code>gss_duplicate_oid</code>	38
<code>gss_acquire_cred</code>	10	<code>gss_encapsulate_token</code>	38
<code>gss_add_cred</code>	11	<code>GSS_ERROR</code>	9
<code>gss_add_oid_set_member</code>	34	<code>gss_export_name</code>	32
<code>GSS_C_ANON_FLAG</code>	20, 25	<code>gss_import_name</code>	29
<code>GSS_C_CONF_FLAG</code>	20, 25	<code>gss_indicate_mechs</code>	35
<code>GSS_C_DELEG_FLAG</code>	19, 20, 24	<code>gss_init_sec_context</code>	16
<code>GSS_C_INTEG_FLAG</code>	20, 25	<code>gss_inquire_cred</code>	14
<code>GSS_C_MUTUAL_FLAG</code>	19, 20, 24	<code>gss_inquire_cred_by_mech</code>	15
<code>GSS_C_PROT_READY_FLAG</code>	20, 25	<code>gss_inquire_mechs_for_name</code>	31
<code>GSS_C_REPLAY_FLAG</code>	19, 20, 25	<code>gss_inquire_names_for_mech</code>	31
<code>GSS_C_SEQUENCE_FLAG</code>	20, 25	<code>gss_oid_equal</code>	38
<code>GSS_C_TRANS_FLAG</code>	21, 25	<code>gss_release_buffer</code>	33
<code>GSS_CALLING_ERROR</code>	9	<code>gss_release_cred</code>	15
<code>gss_canonicalize_name</code>	31, 32	<code>gss_release_name</code>	30
<code>gss_check_version</code>	38	<code>gss_release_oid_set</code>	35
<code>gss_compare_name</code>	30	<code>GSS_ROUTINE_ERROR</code>	9
<code>gss_copy_oid</code>	38	<code>GSS_S</code>	7
<code>gss_create_empty_oid_set</code>	34	<code>GSS_SUPPLEMENTARY_INFO</code>	9
<code>gss_decapsulate_token</code>	38	<code>gss_test_oid_set_member</code>	34
<code>gss_delete_sec_context</code>	26	<code>gss_unwrap</code>	28
<code>gss_display_name</code>	29	<code>gss_wrap</code>	27
<code>gss_display_status</code>	35		
<code>gss_duplicate_name</code>	33		

X

<code>xalloc_fail_func</code>	6
-------------------------------------	---

Short Contents

1	Introduction	1
2	Preparation	4
3	Standard GSS API	7
4	Extended GSS API	38
5	Acknowledgements	39
A	Criticism of GSS	40
	Concept Index	42
	API Index	43

Table of Contents

1	Introduction	1
1.1	Getting Started	1
1.2	Features	1
1.3	Supported Platforms	1
1.4	Bug Reports	3
2	Preparation	4
2.1	Header	4
2.2	Initialization	4
2.3	Version Check	4
2.4	Building the source	5
2.5	Out of Memory handling	5
3	Standard GSS API	7
3.1	Error Handling	7
3.1.1	GSS status codes	7
3.1.2	Mechanism-specific status codes	9
3.2	Credential Management	9
3.3	Context-Level Routines	15
3.4	Per-Message Routines	26
3.5	Name Manipulation	28
3.6	Miscellaneous Routines	33
4	Extended GSS API	38
5	Acknowledgements	39
	Appendix A Criticism of GSS	40
	Concept Index	42
	API Index	43