           Generic Security Service Application Program Interface

Status of this Memo

   This RFC specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
   improvements. Please refer to the current edition of the "Internet
   Official Protocol Standards" for the standardization state and status
   of this protocol. Distribution of this memo is unlimited.

Abstract

   This Generic Security Service Application Program Interface (GSS-API)
   definition provides security services to callers in a generic
   fashion, supportable with a range of underlying mechanisms and
   technologies and hence allowing source-level portability of
   applications to different environments. This specification defines
   GSS-API services and primitives at a level independent of underlying
   mechanism and programming language environment, and is to be
   complemented by other, related specifications:

        documents defining specific parameter bindings for particular
        language environments

        documents defining token formats, protocols, and procedures to
        be implemented in order to realize GSS-API services atop
        particular security mechanisms

Table of Contents

1. GSS-API Characteristics and Concepts

   The operational paradigm in which GSS-API operates is as follows. A
   typical GSS-API caller is itself a communications protocol, calling
   on GSS-API in order to protect its communications with
   authentication, integrity, and/or confidentiality security services.
   A GSS-API caller accepts tokens provided to it by its local GSS-API
   implementation and transfers the tokens to a peer on a remote system;
   that peer passes the received tokens to its local GSS-API
   implementation for processing. The security services available
   through GSS-API in this fashion are implementable (and have been

implemented) over a range of underlying mechanisms based on secret-
key and public-key cryptographic technologies.

The GSS-API separates the operations of initializing a security
context between peers, achieving peer entity authentication (This
security service definition, and other definitions used in this
document, corresponds to that provided in International Standard ISO
7498-2-1988(E), Security Architecture.) (GSS_Init_sec_context() and
GSS_Accept_sec_context() calls), from the operations of providing
per-message data origin authentication and data integrity protection
(GSS_Sign() and GSS_Verify() calls) for messages subsequently
transferred in conjunction with that context. Per-message GSS_Seal()
and GSS_Unseal() calls provide the data origin authentication and
data integrity services which GSS_Sign() and GSS_Verify() offer, and
also support selection of confidentiality services as a caller
option. Additional calls provide supportive functions to the GSS-
API's users.

The following paragraphs provide an example illustrating the
dataflows involved in use of the GSS-API by a client and server in a
mechanism-independent fashion, establishing a security context and
transferring a protected message. The example assumes that credential
acquisition has already been completed. The example assumes that the
underlying authentication technology is capable of authenticating a
client to a server using elements carried within a single token, and
of authenticating the server to the client (mutual authentication)
with a single returned token; this assumption holds for presently-
documented CAT mechanisms but is not necessarily true for other
cryptographic technologies and associated protocols.

The client calls GSS_Init_sec_context() to establish a security
context to the server identified by targ_name, and elects to set the
mutual_req_flag so that mutual authentication is performed in the
course of context establishment. GSS_Init_sec_context() returns an
output_token to be passed to the server, and indicates
GSS_CONTINUE_NEEDED status pending completion of the mutual
authentication sequence. Had mutual_req_flag not been set, the
initial call to GSS_Init_sec_context() would have returned
GSS_COMPLETE status. The client sends the output_token to the server.

The server passes the received token as the input_token parameter to
GSS_Accept_sec_context(). GSS_Accept_sec_context indicates
GSS_COMPLETE status, provides the client's authenticated identity in
the src_name result, and provides an output_token to be passed to the
client. The server sends the output_token to the client.

The client passes the received token as the input_token parameter to
a successor call to GSS_Init_sec_context(), which processes data

included in the token in order to achieve mutual authentication from
the client's viewpoint. This call to GSS_Init_sec_context() returns
GSS_COMPLETE status, indicating successful mutual authentication and
the completion of context establishment for this example.

The client generates a data message and passes it to GSS_Seal().
GSS_Seal() performs data origin authentication, data integrity, and
(optionally) confidentiality processing on the message and
encapsulates the result into output_message, indicating GSS_COMPLETE
status. The client sends the output_message to the server.

The server passes the received message to GSS_Unseal(). GSS_Unseal
inverts the encapsulation performed by GSS_Seal(), deciphers the
message if the optional confidentiality feature was applied, and
validates the data origin authentication and data integrity checking
quantities. GSS_Unseal() indicates successful validation by
returning GSS_COMPLETE status along with the resultant
output_message.

For purposes of this example, we assume that the server knows by
out-of-band means that this context will have no further use after
one protected message is transferred from client to server. Given
this premise, the server now calls GSS_Delete_sec_context() to flush
context-level information. GSS_Delete_sec_context() returns a
context_token for the server to pass to the client.

The client passes the returned context_token to
GSS_Process_context_token(), which returns GSS_COMPLETE status after
deleting context-level information at the client system.

The GSS-API design assumes and addresses several basic goals,
including:

   Mechanism independence: The GSS-API defines an interface to
   cryptographically implemented strong authentication and other
   security services at a generic level which is independent of
   particular underlying mechanisms. For example, GSS-API-provided
   services can be implemented by secret-key technologies (e.g.,
   Kerberos) or public-key approaches (e.g., X.509).

   Protocol environment independence: The GSS-API is independent of
   the communications protocol suites with which it is employed,
   permitting use in a broad range of protocol environments. In
   appropriate environments, an intermediate implementation "veneer"
   which is oriented to a particular communication protocol (e.g.,
   Remote Procedure Call (RPC)) may be interposed between
   applications which call that protocol and the GSS-API, thereby
   invoking GSS-API facilities in conjunction with that protocol's

communications invocations.

Protocol association independence: The GSS-API's security context
construct is independent of communications protocol association
constructs. This characteristic allows a single GSS-API
implementation to be utilized by a variety of invoking protocol
modules on behalf of those modules' calling applications. GSS-API
services can also be invoked directly by applications, wholly
independent of protocol associations.

Suitability to a range of implementation placements: GSS-API
clients are not constrained to reside within any Trusted Computing
Base (TCB) perimeter defined on a system where the GSS-API is
implemented; security services are specified in a manner suitable
to both intra-TCB and extra-TCB callers.

1.1. GSS-API Constructs

This section describes the basic elements comprising the GSS-API.

1.1.1. Credentials

Credentials structures provide the prerequisites enabling peers to
establish security contexts with each other. A caller may designate
that its default credential be used for context establishment calls
without presenting an explicit handle to that credential.
Alternately, those GSS-API callers which need to make explicit
selection of particular credentials structures may make references to
those credentials through GSS-API-provided credential handles
("cred_handles").

A single credential structure may be used for initiation of outbound
contexts and acceptance of inbound contexts. Callers needing to
operate in only one of these modes may designate this fact when
credentials are acquired for use, allowing underlying mechanisms to
optimize their processing and storage requirements. The credential
elements defined by a particular mechanism may contain multiple
cryptographic keys, e.g., to enable authentication and message
encryption to be performed with different algorithms.

A single credential structure may accommodate credential information
associated with multiple underlying mechanisms (mech_types); a
credential structure's contents will vary depending on the set of
mech_types supported by a particular GSS-API implementation.
Commonly, a single mech_type will be used for all security contexts
established by a particular initiator to a particular target; the
primary motivation for supporting credential sets representing
multiple mech_types is to allow initiators on systems which are

equipped to handle multiple types to initiate contexts to targets on
other systems which can accommodate only a subset of the set
supported at the initiator's system.

It is the responsibility of underlying system-specific mechanisms and
OS functions below the GSS-API to ensure that the ability to acquire
and use credentials associated with a given identity is constrained
to appropriate processes within a system. This responsibility should
be taken seriously by implementors, as the ability for an entity to
utilize a principal's credentials is equivalent to the entity's
ability to successfully assert that principal's identity.

Once a set of GSS-API credentials is established, the transferability
of that credentials set to other processes or analogous constructs
within a system is a local matter, not defined by the GSS-API. An
example local policy would be one in which any credentials received
as a result of login to a given user account, or of delegation of
rights to that account, are accessible by, or transferable to,
processes running under that account.

The credential establishment process (particularly when performed on
behalf of users rather than server processes) is likely to require
access to passwords or other quantities which should be protected
locally and exposed for the shortest time possible. As a result, it
will often be appropriate for preliminary credential establishment to
be performed through local means at user login time, with the
result(s) cached for subsequent reference. These preliminary
credentials would be set aside (in a system-specific fashion) for
subsequent use, either:

    to be accessed by an invocation of the GSS-API GSS_Acquire_cred()
    call, returning an explicit handle to reference that credential

    as the default credentials installed on behalf of a process

1.1.2. Tokens

   Tokens are data elements transferred between GSS-API callers, and are
   divided into two classes. Context-level tokens are exchanged in order
   to establish and manage a security context between peers. Per-message
   tokens are exchanged in conjunction with an established context to
   provide protective security services for corresponding data messages.
   The internal contents of both classes of tokens are specific to the
   particular underlying mechanism used to support the GSS-API; Appendix
   B of this document provides a uniform recommendation for designers of
   GSS-API support mechanisms, encapsulating mechanism-specific
   information along with a globally-interpretable mechanism identifier.

Tokens are opaque from the viewpoint of GSS-API callers. They are
generated within the GSS-API implementation at an end system,
provided to a GSS-API caller to be transferred to the peer GSS-API
caller at a remote end system, and processed by the GSS-API
implementation at that remote end system. Tokens may be output by
GSS-API primitives (and are to be transferred to GSS-API peers)
independent of the status indications which those primitives
indicate. Token transfer may take place in an in-band manner,
integrated into the same protocol stream used by the GSS-API callers
for other data transfers, or in an out-of-band manner across a
logically separate channel.

Development of GSS-API support primitives based on a particular
underlying cryptographic technique and protocol does not necessarily
imply that GSS-API callers invoking that GSS-API mechanism type will
be able to interoperate with peers invoking the same technique and
protocol outside the GSS-API paradigm. For example, the format of
GSS-API tokens defined in conjunction with a particular mechanism,
and the techniques used to integrate those tokens into callers'
protocols, may not be the same as those used by non-GSS-API callers
of the same underlying technique.

1.1.3. Security Contexts

Security contexts are established between peers, using credentials
established locally in conjunction with each peer or received by
peers via delegation. Multiple contexts may exist simultaneously
between a pair of peers, using the same or different sets of
credentials. Coexistence of multiple contexts using different
credentials allows graceful rollover when credentials expire.
Distinction among multiple contexts based on the same credentials
serves applications by distinguishing different message streams in a
security sense.

The GSS-API is independent of underlying protocols and addressing
structure, and depends on its callers to transport GSS-API-provided
data elements. As a result of these factors, it is a caller
responsibility to parse communicated messages, separating GSS-API-
related data elements from caller-provided data. The GSS-API is
independent of connection vs. connectionless orientation of the
underlying communications service.

No correlation between security context and communications protocol
association is dictated. (The optional channel binding facility,
discussed in Section 1.1.6 of this document, represents an
intentional exception to this rule, supporting additional protection
features within GSS-API supporting mechanisms.) This separation
allows the GSS-API to be used in a wide range of communications

environments, and also simplifies the calling sequences of the
individual calls. In many cases (depending on underlying security
protocol, associated mechanism, and availability of cached
information), the state information required for context setup can be
sent concurrently with initial signed user data, without interposing
additional message exchanges.

1.1.4. Mechanism Types

   In order to successfully establish a security context with a target
   peer, it is necessary to identify an appropriate underlying mechanism
   type (mech_type) which both initiator and target peers support. The
   definition of a mechanism embodies not only the use of a particular
   cryptographic technology (or a hybrid or choice among alternative
   cryptographic technologies), but also definition of the syntax and
   semantics of data element exchanges which that mechanism will employ
   in order to support security services.

   It is recommended that callers initiating contexts specify the
   "default" mech_type value, allowing system-specific functions within
   or invoked by the GSS-API implementation to select the appropriate
   mech_type, but callers may direct that a particular mech_type be
   employed when necessary.

   The means for identifying a shared mech_type to establish a security
   context with a peer will vary in different environments and
   circumstances; examples include (but are not limited to):

      use of a fixed mech_type, defined by configuration, within an
      environment

      syntactic convention on a target-specific basis, through
      examination of a target's name

      lookup of a target's name in a naming service or other database in
      order to identify mech_types supported by that target

      explicit negotiation between GSS-API callers in advance of
      security context setup

   When transferred between GSS-API peers, mech_type specifiers (per
   Appendix B, represented as Object Identifiers (OIDs)) serve to
   qualify the interpretation of associated tokens. (The structure and
   encoding of Object Identifiers is defined in ISO/IEC 8824,
   "Specification of Abstract Syntax Notation One (ASN.1)" and in
   ISO/IEC 8825, "Specification of Basic Encoding Rules for Abstract
   Syntax Notation One (ASN.1)".) Use of hierarchically structured OIDs
   serves to preclude ambiguous interpretation of mech_type specifiers.

The OID representing the DASS MechType, for example, is
1.3.12.2.1011.7.5.

1.1.5. Naming

The GSS-API avoids prescription of naming structures, treating the
names transferred across the interface in order to initiate and
accept security contexts as opaque octet string quantities. This
approach supports the GSS-API's goal of implementability atop a range
of underlying security mechanisms, recognizing the fact that
different mechanisms process and authenticate names which are
presented in different forms. Generalized services offering
translation functions among arbitrary sets of naming environments are
outside the scope of the GSS-API; availability and use of local
conversion functions to translate among the naming formats supported
within a given end system is anticipated.

Two distinct classes of name representations are used in conjunction
with different GSS-API parameters:

    a printable form (denoted by OCTET STRING), for acceptance from
    and presentation to users; printable name forms are accompanied by
    OID tags identifying the namespace to which they correspond

    an internal form (denoted by INTERNAL NAME), opaque to callers and
    defined by individual GSS-API implementations; GSS-API
    implementations supporting multiple namespace types are
    responsible for maintaining internal tags to disambiguate the
    interpretation of particular names

    Tagging of printable names allows GSS-API callers and underlying
    GSS-API mechanisms to disambiguate name types and to determine
    whether an associated name's type is one which they are capable of
    processing, avoiding aliasing problems which could result from
    misinterpreting a name of one type as a name of another type.

In addition to providing means for names to be tagged with types,
this specification defines primitives to support a level of naming
environment independence for certain calling applications. To provide
basic services oriented towards the requirements of callers which
need not themselves interpret the internal syntax and semantics of
names, GSS-API calls for name comparison (GSS_Compare_name()),
human-readable display (GSS_Display_name()), input conversion
(GSS_Import_name()), and internal name deallocation
(GSS_Release_name()) functions are defined. (It is anticipated that
these proposed GSS-API calls will be implemented in many end systems
based on system-specific name manipulation primitives already extant
within those end systems; inclusion within the GSS-API is intended to

offer GSS-API callers a portable means to perform specific
operations, supportive of authorization and audit requirements, on
authenticated names.)

GSS_Import_name() implementations can, where appropriate, support
more than one printable syntax corresponding to a given namespace
(e.g., alternative printable representations for X.500 Distinguished
Names), allowing flexibility for their callers to select among
alternative representations. GSS_Display_name() implementations
output a printable syntax selected as appropriate to their
operational environments; this selection is a local matter. Callers
desiring portability across alternative printable syntaxes should
refrain from implementing comparisons based on printable name forms
and should instead use the GSS_Compare_name() call to determine
whether or not one internal-format name matches another.

1.1.6. Channel Bindings

The GSS-API accommodates the concept of caller-provided channel
binding ("chan_binding") information, used by GSS-API callers to bind
the establishment of a security context to relevant characteristics
(e.g., addresses, transformed representations of encryption keys) of
the underlying communications channel and of protection mechanisms
applied to that communications channel. Verification by one peer of
chan_binding information provided by the other peer to a context
serves to protect against various active attacks. The caller
initiating a security context must determine the chan_binding values
before making the GSS_Init_sec_context() call, and consistent values
must be provided by both peers to a context. Callers should not
assume that underlying mechanisms provide confidentiality protection
for channel binding information.

Use or non-use of the GSS-API channel binding facility is a caller
option, and GSS-API supporting mechanisms can support operation in an
environment where NULL channel bindings are presented. When non-NULL
channel bindings are used, certain mechanisms will offer enhanced
security value by interpreting the bindings' content (rather than
simply representing those bindings, or signatures computed on them,
within tokens) and will therefore depend on presentation of specific
data in a defined format. To this end, agreements among mechanism
implementors are defining conventional interpretations for the
contents of channel binding arguments, including address specifiers
(with content dependent on communications protocol environment) for
context initiators and acceptors. (These conventions are being
incorporated into related documents.) In order for GSS-API callers to
be portable across multiple mechanisms and achieve the full security
functionality available from each mechanism, it is strongly
recommended that GSS-API callers provide channel bindings consistent

with these conventions and those of the networking environment in
which they operate.

1.2. GSS-API Features and Issues

This section describes aspects of GSS-API operations, of the security
services which the GSS-API provides, and provides commentary on
design issues.

1.2.1. Status Reporting

Each GSS-API call provides two status return values. Major_status
values provide a mechanism-independent indication of call status
(e.g., GSS_COMPLETE, GSS_FAILURE, GSS_CONTINUE_NEEDED), sufficient to
drive normal control flow within the caller in a generic fashion.
Table 1 summarizes the defined major_status return codes in tabular
fashion.

Table 1: GSS-API Major Status Codes

     FATAL ERROR CODES

     GSS_BAD_BINDINGS   channel binding mismatch
     GSS_BAD_MECH   unsupported mechanism requested
     GSS_BAD_NAME   invalid name provided
     GSS_BAD_NAMETYPE   name of unsupported type provided
     GSS_BAD_STATUS   invalid input status selector
     GSS_BAD_SIG   token had invalid signature
     GSS_CONTEXT_EXPIRED   specified security context expired
     GSS_CREDENTIALS_EXPIRED   expired credentials detected
     GSS_DEFECTIVE_CREDENTIAL   defective credential detected
     GSS_DEFECTIVE_TOKEN   defective token detected
     GSS_FAILURE   failure, unspecified at GSS-API
                                level
     GSS_NO_CONTEXT   no valid security context specified
     GSS_NO_CRED   no valid credentials provided

     INFORMATORY STATUS CODES

     GSS_COMPLETE   normal completion
     GSS_CONTINUE_NEEDED   continuation call to routine
                                required
     GSS_DUPLICATE_TOKEN   duplicate per-message token
                                detected
     GSS_OLD_TOKEN   timed-out per-message token
                                detected
     GSS_UNSEQ_TOKEN   out-of-order per-message token
                                detected

Minor_status provides more detailed status information which may
include status codes specific to the underlying security mechanism.
Minor_status values are not specified in this document.

GSS_CONTINUE_NEEDED major_status returns, and optional message
outputs, are provided in GSS_Init_sec_context() and
GSS_Accept_sec_context() calls so that different mechanisms'
employment of different numbers of messages within their
authentication sequences need not be reflected in separate code paths
within calling applications. Instead, such cases are accomodated with
sequences of continuation calls to GSS_Init_sec_context() and
GSS_Accept_sec_context(). The same mechanism is used to encapsulate
mutual authentication within the GSS-API's context initiation calls.

For mech_types which require interactions with third-party servers in
order to establish a security context, GSS-API context establishment
calls may block pending completion of such third-party interactions.
On the other hand, no GSS-API calls pend on serialized interactions
with GSS-API peer entities. As a result, local GSS-API status
returns cannot reflect unpredictable or asynchronous exceptions
occurring at remote peers, and reflection of such status information
is a caller responsibility outside the GSS-API.

1.2.2. Per-Message Security Service Availability

When a context is established, two flags are returned to indicate the
set of per-message protection security services which will be
available on the context:

    the integ_avail flag indicates whether per-message integrity and
    data origin authentication services are available

    the conf_avail flag indicates whether per-message confidentiality
    services are available, and will never be returned TRUE unless the
    integ_avail flag is also returned TRUE

    GSS-API callers desiring per-message security services should
    check the values of these flags at context establishment time, and
    must be aware that a returned FALSE value for integ_avail means
    that invocation of GSS_Sign() or GSS_Seal() primitives on the
    associated context will apply no cryptographic protection to user
    data messages.

The GSS-API per-message protection service primitives, as the
category name implies, are oriented to operation at the granularity
of protocol data units. They perform cryptographic operations on the
data units, transfer cryptographic control information in tokens,
and, in the case of GSS_Seal(), encapsulate the protected data unit.

As such, these primitives are not oriented to efficient data
protection for stream-paradigm protocols (e.g., Telnet) if
cryptography must be applied on an octet-by-octet basis.

1.2.3. Per-Message Replay Detection and Sequencing

Certain underlying mech_types are expected to offer support for
replay detection and/or sequencing of messages transferred on the
contexts they support. These optionally-selectable protection
features are distinct from replay detection and sequencing features
applied to the context establishment operation itself; the presence
or absence of context-level replay or sequencing features is wholly a
function of the underlying mech_type's capabilities, and is not
selected or omitted as a caller option.

The caller initiating a context provides flags (replay_det_req_flag
and sequence_req_flag) to specify whether the use of per-message
replay detection and sequencing features is desired on the context
being established. The GSS-API implementation at the initiator system
can determine whether these features are supported (and whether they
are optionally selectable) as a function of mech_type, without need
for bilateral negotiation with the target. When enabled, these
features provide recipients with indicators as a result of GSS-API
processing of incoming messages, identifying whether those messages
were detected as duplicates or out-of-sequence. Detection of such
events does not prevent a suspect message from being provided to a
recipient; the appropriate course of action on a suspect message is a
matter of caller policy.

The semantics of the replay detection and sequencing services applied
to received messages, as visible across the interface which the GSS-
API provides to its clients, are as follows:

When replay_det_state is TRUE, the possible major_status returns for
well-formed and correctly signed messages are as follows:

    1. GSS_COMPLETE indicates that the message was within the window
    (of time or sequence space) allowing replay events to be detected,
    and that the message was not a replay of a previously-processed
    message within that window.

    2. GSS_DUPLICATE_TOKEN indicates that the signature on the
    received message was correct, but that the message was recognized
    as a duplicate of a previously-processed message.

    3. GSS_OLD_TOKEN indicates that the signature on the received
    message was correct, but that the message is too old to be checked
    for duplication.

When sequence_state is TRUE, the possible major_status returns for
well-formed and correctly signed messages are as follows:

   1. GSS_COMPLETE indicates that the message was within the window
   (of time or sequence space) allowing replay events to be detected,
   and that the message was not a replay of a previously-processed
   message within that window.

   2. GSS_DUPLICATE_TOKEN indicates that the signature on the
   received message was correct, but that the message was recognized
   as a duplicate of a previously-processed message.

   3. GSS_OLD_TOKEN indicates that the signature on the received
   message was correct, but that the token is too old to be checked
   for duplication.

   4. GSS_UNSEQ_TOKEN indicates that the signature on the received
   message was correct, but that it is earlier in a sequenced stream
   than a message already processed on the context. [Note:
   Mechanisms can be architected to provide a stricter form of
   sequencing service, delivering particular messages to recipients
   only after all predecessor messages in an ordered stream have been
   delivered. This type of support is incompatible with the GSS-API
   paradigm in which recipients receive all messages, whether in
   order or not, and provide them (one at a time, without intra-GSS-
   API message buffering) to GSS-API routines for validation. GSS-
   API facilities provide supportive functions, aiding clients to
   achieve strict message stream integrity in an efficient manner in
   conjunction with sequencing provisions in communications
   protocols, but the GSS-API does not offer this level of message
   stream integrity service by itself.]

As the message stream integrity features (especially sequencing) may
interfere with certain applications' intended communications
paradigms, and since support for such features is likely to be
resource intensive, it is highly recommended that mech_types
supporting these features allow them to be activated selectively on
initiator request when a context is established. A context initiator
and target are provided with corresponding indicators
(replay_det_state and sequence_state), signifying whether these
features are active on a given context.

An example mech_type supporting per-message replay detection could
(when replay_det_state is TRUE) implement the feature as follows: The
underlying mechanism would insert timestamps in data elements output
by GSS_Sign() and GSS_Seal(), and would maintain (within a time-
limited window) a cache (qualified by originator-recipient pair)
identifying received data elements processed by GSS_Verify() and

GSS_Unseal(). When this feature is active, exception status returns
(GSS_DUPLICATE_TOKEN, GSS_ OLD_TOKEN) will be provided when
GSS_Verify() or GSS_Unseal() is presented with a message which is
either a detected duplicate of a prior message or which is too old to
validate against a cache of recently received messages.

1.2.4. Quality of Protection

Some mech_types will provide their users with fine granularity
control over the means used to provide per-message protection,
allowing callers to trade off security processing overhead
dynamically against the protection requirements of particular
messages. A per-message quality-of-protection parameter (analogous to
quality-of-service, or QOS) selects among different QOP options
supported by that mechanism. On context establishment for a multi-QOP
mech_type, context-level data provides the prerequisite data for a
range of protection qualities.

It is expected that the majority of callers will not wish to exert
explicit mechanism-specific QOP control and will therefore request
selection of a default QOP. Definitions of, and choices among, non-
default QOP values are mechanism-specific, and no ordered sequences
of QOP values can be assumed equivalent across different mechanisms.
Meaningful use of non-default QOP values demands that callers be
familiar with the QOP definitions of an underlying mechanism or
mechanisms, and is therefore a non-portable construct.

2. Interface Descriptions

This section describes the GSS-API's service interface, dividing the
set of calls offered into four groups. Credential management calls
are related to the acquisition and release of credentials by
principals. Context-level calls are related to the management of
security contexts between principals. Per-message calls are related
to the protection of individual messages on established security
contexts. Support calls provide ancillary functions useful to GSS-API
callers. Table 2 groups and summarizes the calls in tabular fashion.

Table 2: GSS-API Calls

CREDENTIAL MANAGEMENT

GSS_Acquire_cred   acquire credentials for use
GSS_Release_cred   release credentials after use
GSS_Inquire_cred   display information about
                             credentials

CONTEXT-LEVEL CALLS

GSS_Init_sec_context   initiate outbound security context
GSS_Accept_sec_context   accept inbound security context
GSS_Delete_sec_context   flush context when no longer needed
GSS_Process_context_token   process received control token on
                             context
GSS_Context_time   indicate validity time remaining on
                             context

PER-MESSAGE CALLS

GSS_Sign   apply signature, receive as token
                             separate from message
GSS_Verify   validate signature token along with
                             message
GSS_Seal   sign, optionally encrypt,
                             encapsulate
GSS_Unseal   decapsulate, decrypt if needed,
                             validate signature

SUPPORT CALLS

GSS_Display_status   translate status codes to printable
                             form
GSS_Indicate_mechs   indicate mech_types supported on
                             local system
GSS_Compare_name   compare two names for equality
GSS_Display_name   translate name to printable form
GSS_Import_name   convert printable name to
                             normalized form
GSS_Release_name   free storage of normalized-form
                             name
GSS_Release_buffer   free storage of printable name
GSS_Release_oid_set   free storage of OID set object

2.1. Credential management calls

   These GSS-API calls provide functions related to the management of
   credentials. Their characterization with regard to whether or not
   they may block pending exchanges with other network entities (e.g.,
   directories or authentication servers) depends in part on OS-specific
   (extra-GSS-API) issues, so is not specified in this document.

   The GSS_Acquire_cred() call is defined within the GSS-API in support
   of application portability, with a particular orientation towards
   support of portable server applications. It is recognized that (for
   certain systems and mechanisms) credentials for interactive users may
   be managed differently from credentials for server processes; in such
   environments, it is the GSS-API implementation's responsibility to
   distinguish these cases and the procedures for making this
   distinction are a local matter. The GSS_Release_cred() call provides
   a means for callers to indicate to the GSS-API that use of a
   credentials structure is no longer required. The GSS_Inquire_cred()
   call allows callers to determine information about a credentials
   structure.

2.1.1. GSS_Acquire_cred call

   Inputs:

   o desired_name INTERNAL NAME, -NULL requests locally-determined
      default

   o lifetime_req INTEGER,-in seconds; 0 requests default

   o desired_mechs SET OF OBJECT IDENTIFIER,-empty set requests
      system-selected default

   o cred_usage INTEGER-0=INITIATE-AND-ACCEPT, 1=INITIATE-ONLY,
      2=ACCEPT-ONLY

   Outputs:

   o major_status INTEGER,

   o minor_status INTEGER,

   o output_cred_handle OCTET STRING,

   o actual_mechs SET OF OBJECT IDENTIFIER,

   o lifetime_rec INTEGER -in seconds, or reserved value for
      INDEFINITE

Return major_status codes:

o GSS_COMPLETE indicates that requested credentials were
   successfully established, for the duration indicated in
   lifetime_rec, suitable for the usage requested in cred_usage, for
   the set of mech_types indicated in actual_mechs, and that those
   credentials can be referenced for subsequent use with the handle
   returned in output_cred_handle.

o GSS_BAD_MECH indicates that a mech_type unsupported by the GSS-API
   implementation type was requested, causing the credential
   establishment operation to fail.

o GSS_BAD_NAMETYPE indicates that the provided desired_name is
   uninterpretable or of a type unsupported by the supporting GSS-API
   implementation, so no credentials could be established for the
   accompanying desired_name.

o GSS_BAD_NAME indicates that the provided desired_name is
   inconsistent in terms of internally-incorporated type specifier
   information, so no credentials could be established for the
   accompanying desired_name.

o GSS_FAILURE indicates that credential establishment failed for
   reasons unspecified at the GSS-API level, including lack of
   authorization to establish and use credentials associated with the
   identity named in the input desired_name argument.

GSS_Acquire_cred() is used to acquire credentials so that a
principal can (as a function of the input cred_usage parameter)
initiate and/or accept security contexts under the identity
represented by the desired_name input argument. On successful
completion, the returned output_cred_handle result provides a handle
for subsequent references to the acquired credentials. Typically,
single-user client processes using only default credentials for
context establishment purposes will have no need to invoke this call.

A caller may provide the value NULL for desired_name, signifying a
request for credentials corresponding to a default principal
identity. The procedures used by GSS-API implementations to select
the appropriate principal identity in response to this form of
request are local matters. It is possible that multiple pre-
established credentials may exist for the same principal identity
(for example, as a result of multiple user login sessions) when
GSS_Acquire_cred() is called; the means used in such cases to select
a specific credential are local matters. The input lifetime_req
argument to GSS_Acquire_cred() may provide useful information for
local GSS-API implementations to employ in making this disambiguation

in a manner which will best satisfy a caller's intent.

The lifetime_rec result indicates the length of time for which the
acquired credentials will be valid, as an offset from the present. A
mechanism may return a reserved value indicating INDEFINITE if no
constraints on credential lifetime are imposed. A caller of
GSS_Acquire_cred() can request a length of time for which acquired
credentials are to be valid (lifetime_req argument), beginning at the
present, or can request credentials with a default validity interval.
(Requests for postdated credentials are not supported within the
GSS-API.) Certain mechanisms and implementations may bind in
credential validity period specifiers at a point preliminary to
invocation of the GSS_Acquire_cred() call (e.g., in conjunction with
user login procedures). As a result, callers requesting non-default
values for lifetime_req must recognize that such requests cannot
always be honored and must be prepared to accommodate the use of
returned credentials with different lifetimes as indicated in
lifetime_rec.

The caller of GSS_Acquire_cred() can explicitly specify a set of
mech_types which are to be accommodated in the returned credentials
(desired_mechs argument), or can request credentials for a system-
defined default set of mech_types. Selection of the system-specified
default set is recommended in the interests of application
portability. The actual_mechs return value may be interrogated by the
caller to determine the set of mechanisms with which the returned
credentials may be used.

2.1.2. GSS_Release_cred call

Input:

o cred_handle OCTET STRING-NULL specifies default credentials

Outputs:

o major_status INTEGER,

o minor_status INTEGER

Return major_status codes:

o GSS_COMPLETE indicates that the credentials referenced by the
   input cred_handle were released for purposes of subsequent access
   by the caller. The effect on other processes which may be
   authorized shared access to such credentials is a local matter.

   o GSS_NO_CRED indicates that no release operation was performed,
      either because the input cred_handle was invalid or because the
      caller lacks authorization to access the referenced credentials.

   o GSS_FAILURE indicates that the release operation failed for
      reasons unspecified at the GSS-API level.

   Provides a means for a caller to explicitly request that credentials
   be released when their use is no longer required. Note that system-
   specific credential management functions are also likely to exist,
   for example to assure that credentials shared among processes are
   properly deleted when all affected processes terminate, even if no
   explicit release requests are issued by those processes. Given the
   fact that multiple callers are not precluded from gaining authorized
   access to the same credentials, invocation of GSS_Release_cred()
   cannot be assumed to delete a particular set of credentials on a
   system-wide basis.

2.1.3. GSS_Inquire_cred call

      Input:

      o cred_handle OCTET STRING -NULL specifies default credentials

      Outputs:

      o major_status INTEGER,

      o minor_status INTEGER,

      o cred_name INTERNAL NAME,

      o lifetime_rec INTEGER -in seconds, or reserved value for
         INDEFINITE

      o cred_usage INTEGER, -0=INITIATE-AND-ACCEPT, 1=INITIATE-ONLY,
         2=ACCEPT-ONLY

      o mech_set SET OF OBJECT IDENTIFIER

      Return major_status codes:

      o GSS_COMPLETE indicates that the credentials referenced by the
         input cred_handle argument were valid, and that the output
         cred_name, lifetime_rec, and cred_usage values represent,
         respectively, the credentials' associated principal name,
         remaining lifetime, suitable usage modes, and supported
         mechanism types.

o GSS_NO_CRED indicates that no information could be returned
   about the referenced credentials, either because the input
   cred_handle was invalid or because the caller lacks
   authorization to access the referenced credentials.

o GSS_FAILURE indicates that the release operation failed for
   reasons unspecified at the GSS-API level.

The GSS_Inquire_cred() call is defined primarily for the use of
those callers which make use of default credentials rather than
acquiring credentials explicitly with GSS_Acquire_cred(). It enables
callers to determine a credential structure's associated principal
name, remaining validity period, usability for security context
initiation and/or acceptance, and supported mechanisms.

2.2. Context-level calls

This group of calls is devoted to the establishment and management of
security contexts between peers. A context's initiator calls
GSS_Init_sec_context(), resulting in generation of a token which the
caller passes to the target. At the target, that token is passed to
GSS_Accept_sec_context(). Depending on the underlying mech_type and
specified options, additional token exchanges may be performed in the
course of context establishment; such exchanges are accommodated by
GSS_CONTINUE_NEEDED status returns from GSS_Init_sec_context() and
GSS_Accept_sec_context(). Either party to an established context may
invoke GSS_Delete_sec_context() to flush context information when a
context is no longer required. GSS_Process_context_token() is used
to process received tokens carrying context-level control
information. GSS_Context_time() allows a caller to determine the
length of time for which an established context will remain valid.

2.2.1. GSS_Init_sec_context call

Inputs:

o claimant_cred_handle OCTET STRING, -NULL specifies "use
   default"

o input_context_handle INTEGER, -0 specifies "none assigned
   yet"

o targ_name INTERNAL NAME,

o mech_type OBJECT IDENTIFIER, -NULL parameter specifies "use
   default"

o deleg_req_flag BOOLEAN,

o mutual_req_flag BOOLEAN,

o replay_det_req_flag BOOLEAN,

o sequence_req_flag BOOLEAN,

o lifetime_req INTEGER,-0 specifies default lifetime

o chan_bindings OCTET STRING,

o input_token OCTET STRING-NULL or token received from target

Outputs:

o major_status INTEGER,

o minor_status INTEGER,

o output_context_handle INTEGER,

o mech_type OBJECT IDENTIFIER, -actual mechanism always
   indicated, never NULL

o output_token OCTET STRING, -NULL or token to pass to context
   target

o deleg_state BOOLEAN,

o mutual_state BOOLEAN,

o replay_det_state BOOLEAN,

o sequence_state BOOLEAN,

o conf_avail BOOLEAN,

o integ_avail BOOLEAN,

o lifetime_rec INTEGER - in seconds, or reserved value for
   INDEFINITE

This call may block pending network interactions for those mech_types
in which an authentication server or other network entity must be
consulted on behalf of a context initiator in order to generate an
output_token suitable for presentation to a specified target.

Return major_status codes:

   o GSS_COMPLETE indicates that context-level information was
     successfully initialized, and that the returned output_token will
     provide sufficient information for the target to perform per-
     message processing on the newly-established context.

   o GSS_CONTINUE_NEEDED indicates that control information in the
     returned output_token must be sent to the target, and that a reply
     must be received and passed as the input_token argument to a
     continuation call to GSS_Init_sec_context(), before per-message
     processing can be performed in conjunction with this context.

   o GSS_DEFECTIVE_TOKEN indicates that consistency checks performed on
     the input_token failed, preventing further processing from being
     performed based on that token.

   o GSS_DEFECTIVE_CREDENTIAL indicates that consistency checks
     performed on the credential structure referenced by
     claimant_cred_handle failed, preventing further processing from
     being performed using that credential structure.

   o GSS_BAD_SIG indicates that the received input_token contains an
     incorrect signature, so context setup cannot be accomplished.

   o GSS_NO_CRED indicates that no context was established, either
     because the input cred_handle was invalid, because the referenced
     credentials are valid for context acceptor use only, or because
     the caller lacks authorization to access the referenced
     credentials.

   o GSS_CREDENTIALS_EXPIRED indicates that the credentials provided
     through the input claimant_cred_handle argument are no longer
     valid, so context establishment cannot be completed.

   o GSS_BAD_BINDINGS indicates that a mismatch between the caller-
     provided chan_bindings and those extracted from the input_token
     was detected, signifying a security-relevant event and preventing
     context establishment. (This result will be returned by
     GSS_Init_sec_context only for contexts where mutual_state is
     TRUE.)

   o GSS_NO_CONTEXT indicates that no valid context was recognized for
     the input context_handle provided; this major status will be
     returned only for successor calls following GSS_CONTINUE_NEEDED
     status returns.

   o GSS_BAD_NAMETYPE indicates that the provided targ_name is of a
     type uninterpretable or unsupported by the supporting GSS-API
     implementation, so context establishment cannot be completed.

   o GSS_BAD_NAME indicates that the provided targ_name is inconsistent
      in terms of internally-incorporated type specifier information, so
      context establishment cannot be accomplished.

   o GSS_FAILURE indicates that context setup could not be accomplished
      for reasons unspecified at the GSS-API level, and that no
      interface-defined recovery action is available.

This routine is used by a context initiator, and ordinarily emits one
(or, for the case of a multi-step exchange, more than one)
output_token suitable for use by the target within the selected
mech_type's protocol. Using information in the credentials structure
referenced by claimant_cred_handle, GSS_Init_sec_context()
initializes the data structures required to establish a security
context with target targ_name. The claimant_cred_handle must
correspond to the same valid credentials structure on the initial
call to GSS_Init_sec_context() and on any successor calls resulting
from GSS_CONTINUE_NEEDED status returns; different protocol sequences
modeled by the GSS_CONTINUE_NEEDED mechanism will require access to
credentials at different points in the context establishment
sequence.

The input_context_handle argument is 0, specifying "not yet
assigned", on the first GSS_Init_sec_context() call relating to a
given context. That call returns an output_context_handle for future
references to this context. When continuation attempts to
GSS_Init_sec_context() are needed to perform context establishment,
the previously-returned non-zero handle value is entered into the
input_context_handle argument and will be echoed in the returned
output_context_handle argument. On such continuation attempts (and
only on continuation attempts) the input_token value is used, to
provide the token returned from the context's target.

The chan_bindings argument is used by the caller to provide
information binding the security context to security-related
characteristics (e.g., addresses, cryptographic keys) of the
underlying communications channel. See Section 1.1.6 of this document
for more discussion of this argument's usage.

The input_token argument contains a message received from the target,
and is significant only on a call to GSS_Init_sec_context() which
follows a previous return indicating GSS_CONTINUE_NEEDED
major_status.

It is the caller's responsibility to establish a communications path
to the target, and to transmit any returned output_token (independent
of the accompanying returned major_status value) to the target over
that path. The output_token can, however, be transmitted along with

the first application-provided input message to be processed by
GSS_Sign() or GSS_Seal() in conjunction with a successfully-
established context.

The initiator may request various context-level functions through
input flags: the deleg_req_flag requests delegation of access rights,
the mutual_req_flag requests mutual authentication, the
replay_det_req_flag requests that replay detection features be
applied to messages transferred on the established context, and the
sequence_req_flag requests that sequencing be enforced. (See Section
1.2.3 for more information on replay detection and sequencing
features.)

Not all of the optionally-requestable features will be available in
all underlying mech_types; the corresponding return state values
(deleg_state, mutual_state, replay_det_state, sequence_state)
indicate, as a function of mech_type processing capabilities and
initiator-provided input flags, the set of features which will be
active on the context. These state indicators' values are undefined
unless the routine's major_status indicates COMPLETE. Failure to
provide the precise set of features requested by the caller does not
cause context establishment to fail; it is the caller's prerogative
to delete the context if the feature set provided is unsuitable for
the caller's use. The returned mech_type value indicates the
specific mechanism employed on the context, and will never indicate
the value for "default".

The conf_avail return value indicates whether the context supports
per-message confidentiality services, and so informs the caller
whether or not a request for encryption through the conf_req_flag
input to GSS_Seal() can be honored. In similar fashion, the
integ_avail return value indicates whether per-message integrity
services are available (through either GSS_Sign() or GSS_Seal()) on
the established context.

The lifetime_req input specifies a desired upper bound for the
lifetime of the context to be established, with a value of 0 used to
request a default lifetime. The lifetime_rec return value indicates
the length of time for which the context will be valid, expressed as
an offset from the present; depending on mechanism capabilities,
credential lifetimes, and local policy, it may not correspond to the
value requested in lifetime_req. If no constraints on context
lifetime are imposed, this may be indicated by returning a reserved
value representing INDEFINITE lifetime_req. The values of conf_avail,
integ_avail, and lifetime_rec are undefined unless the routine's
major_status indicates COMPLETE.

If the mutual_state is TRUE, this fact will be reflected within the

output_token. A call to GSS_Accept_sec_context() at the target in
conjunction with such a context will return a token, to be processed
by a continuation call to GSS_Init_sec_context(), in order to achieve
mutual authentication.

2.2.2. GSS_Accept_sec_context call

Inputs:

o   acceptor_cred_handle OCTET STRING,-NULL specifies "use
    default"

o   input_context_handle INTEGER, -0 specifies "not yet assigned"

o   chan_bindings OCTET STRING,

o   input_token OCTET STRING

Outputs:

o   major_status INTEGER,

o   minor_status INTEGER,

o   src_name INTERNAL NAME,

o   mech_type OBJECT IDENTIFIER,

o   output_context_handle INTEGER,

o   deleg_state BOOLEAN,

o   mutual_state BOOLEAN,

o   replay_det_state BOOLEAN,

o   sequence_state BOOLEAN,

o   conf_avail BOOLEAN,

o   integ_avail BOOLEAN,

o   lifetime_rec INTEGER, - in seconds, or reserved value for
    INDEFINITE

o   delegated_cred_handle OCTET STRING,

o   output_token OCTET STRING -NULL or token to pass to context

initiator

This call may block pending network interactions for those mech_types
in which a directory service or other network entity must be
consulted on behalf of a context acceptor in order to validate a
received input_token.

Return major_status codes:

o GSS_COMPLETE indicates that context-level data structures were
   successfully initialized, and that per-message processing can now
   be performed in conjunction with this context.

o GSS_CONTINUE_NEEDED indicates that control information in the
   returned output_token must be sent to the initiator, and that a
   response must be received and passed as the input_token argument
   to a continuation call to GSS_Accept_sec_context(), before per-
   message processing can be performed in conjunction with this
   context.

o GSS_DEFECTIVE_TOKEN indicates that consistency checks performed on
   the input_token failed, preventing further processing from being
   performed based on that token.

o GSS_DEFECTIVE_CREDENTIAL indicates that consistency checks
   performed on the credential structure referenced by
   acceptor_cred_handle failed, preventing further processing from
   being performed using that credential structure.

o GSS_BAD_SIG indicates that the received input_token contains an
   incorrect signature, so context setup cannot be accomplished.

o GSS_DUPLICATE_TOKEN indicates that the signature on the received
   input_token was correct, but that the input_token was recognized
   as a duplicate of an input_token already processed. No new context
   is established.

o GSS_OLD_TOKEN indicates that the signature on the received
   input_token was correct, but that the input_token is too old to be
   checked for duplication against previously-processed input_tokens.
   No new context is established.

o GSS_NO_CRED indicates that no context was established, either
   because the input cred_handle was invalid, because the referenced
   credentials are valid for context initiator use only, or because
   the caller lacks authorization to access the referenced
   credentials.

o GSS_CREDENTIALS_EXPIRED indicates that the credentials provided
   through the input acceptor_cred_handle argument are no longer
   valid, so context establishment cannot be completed.

o GSS_BAD_BINDINGS indicates that a mismatch between the caller-
   provided chan_bindings and those extracted from the input_token
   was detected, signifying a security-relevant event and preventing
   context establishment.

o GSS_NO_CONTEXT indicates that no valid context was recognized for
   the input context_handle provided; this major status will be
   returned only for successor calls following GSS_CONTINUE_NEEDED
   status returns.

o GSS_FAILURE indicates that context setup could not be accomplished
   for reasons unspecified at the GSS-API level, and that no
   interface-defined recovery action is available.

The GSS_Accept_sec_context() routine is used by a context target.
Using information in the credentials structure referenced by the
input acceptor_cred_handle, it verifies the incoming input_token and
(following the successful completion of a context establishment
sequence) returns the authenticated src_name and the mech_type used.
The acceptor_cred_handle must correspond to the same valid
credentials structure on the initial call to GSS_Accept_sec_context()
and on any successor calls resulting from GSS_CONTINUE_NEEDED status
returns; different protocol sequences modeled by the
GSS_CONTINUE_NEEDED mechanism will require access to credentials at
different points in the context establishment sequence.

The input_context_handle argument is 0, specifying "not yet
assigned", on the first GSS_Accept_sec_context() call relating to a
given context. That call returns an output_context_handle for future
references to this context; when continuation attempts to
GSS_Accept_sec_context() are needed to perform context
establishment, that handle value will be entered into the
input_context_handle argument.

The chan_bindings argument is used by the caller to provide
information binding the security context to security-related
characteristics (e.g., addresses, cryptographic keys) of the
underlying communications channel. See Section 1.1.6 of this document
for more discussion of this argument's usage.

The returned state results (deleg_state, mutual_state,
replay_det_state, and sequence_state) reflect the same context state
values as returned to GSS_Init_sec_context()'s caller at the
initiator system.

The conf_avail return value indicates whether the context supports
per-message confidentiality services, and so informs the caller
whether or not a request for encryption through the conf_req_flag
input to GSS_Seal() can be honored. In similar fashion, the
integ_avail return value indicates whether per-message integrity
services are available (through either GSS_Sign() or GSS_Seal()) on
the established context.

The lifetime_rec return value indicates the length of time for which
the context will be valid, expressed as an offset from the present.
The values of deleg_state, mutual_state, replay_det_state,
sequence_state, conf_avail, integ_avail, and lifetime_rec are
undefined unless the accompanying major_status indicates COMPLETE.

The delegated_cred_handle result is significant only when deleg_state
is TRUE, and provides a means for the target to reference the
delegated credentials. The output_token result, when non-NULL,
provides a context-level token to be returned to the context
initiator to continue a multi-step context establishment sequence. As
noted with GSS_Init_sec_context(), any returned token should be
transferred to the context's peer (in this case, the context
initiator), independent of the value of the accompanying returned
major_status.

Note: A target must be able to distinguish a context-level
input_token, which is passed to GSS_Accept_sec_context(), from the
per-message data elements passed to GSS_Verify() or GSS_Unseal().
These data elements may arrive in a single application message, and
GSS_Accept_sec_context() must be performed before per-message
processing can be performed successfully.

2.2.3. GSS_Delete_sec_context call

   Input:

   o context_handle INTEGER

   Outputs:

   o major_status INTEGER,

   o minor_status INTEGER,

   o output_context_token OCTET STRING

   Return major_status codes:

o GSS_COMPLETE indicates that the context was recognized, that
  relevant context-specific information was flushed, and that the
  returned output_context_token is ready for transfer to the
  context's peer.

o GSS_NO_CONTEXT indicates that no valid context was recognized for
  the input context_handle provide, so no deletion was performed.

o GSS_FAILURE indicates that the context is recognized, but that the
  GSS_Delete_sec_context() operation could not be performed for
  reasons unspecified at the GSS-API level.

This call may block pending network interactions for mech_types in
which active notification must be made to a central server when a
security context is to be deleted.

This call can be made by either peer in a security context, to flush
context-specific information and to return an output_context_token
which can be passed to the context's peer informing it that the
peer's corresponding context information can also be flushed. (Once a
context is established, the peers involved are expected to retain
cached credential and context-related information until the
information's expiration time is reached or until a
GSS_Delete_sec_context() call is made.) Attempts to perform per-
message processing on a deleted context will result in error returns.

2.2.4. GSS_Process_context_token call

   Inputs:

   o context_handle INTEGER,

   o input_context_token OCTET STRING

   Outputs:

   o major_status INTEGER,

   o minor_status INTEGER,

   Return major_status codes:

   o GSS_COMPLETE indicates that the input_context_token was
      successfully processed in conjunction with the context referenced
      by context_handle.

   o GSS_DEFECTIVE_TOKEN indicates that consistency checks performed on
      the received context_token failed, preventing further processing

from being performed with that token.

o GSS_NO_CONTEXT indicates that no valid context was recognized for
   the input context_handle provided.

o GSS_FAILURE indicates that the context is recognized, but that the
   GSS_Process_context_token() operation could not be performed for
   reasons unspecified at the GSS-API level.

This call is used to process context_tokens received from a peer once
a context has been established, with corresponding impact on
context-level state information. One use for this facility is
processing of the context_tokens generated by
GSS_Delete_sec_context(); GSS_Process_context_token() will not block
pending network interactions for that purpose. Another use is to
process tokens indicating remote-peer context establishment failures
after the point where the local GSS-API implementation has already
indicated GSS_COMPLETE status.

2.2.5. GSS_Context_time call

   Input:

   o context_handle INTEGER,

   Outputs:

   o major_status INTEGER,

   o minor_status INTEGER,

   o lifetime_rec INTEGER - in seconds, or reserved value for
      INDEFINITE

   Return major_status codes:

   o GSS_COMPLETE indicates that the referenced context is valid, and
      will remain valid for the amount of time indicated in
      lifetime_rec.

   o GSS_CONTEXT_EXPIRED indicates that data items related to the
      referenced context have expired.

   o GSS_CREDENTIALS_EXPIRED indicates that the context is recognized,
      but that its associated credentials have expired.

   o GSS_NO_CONTEXT indicates that no valid context was recognized for
      the input context_handle provided.

   o GSS_FAILURE indicates that the requested operation failed for
      reasons unspecified at the GSS-API level.

   This call is used to determine the amount of time for which a
   currently established context will remain valid.

2.3. Per-message calls

   This group of calls is used to perform per-message protection
   processing on an established security context. None of these calls
   block pending network interactions. These calls may be invoked by a
   context's initiator or by the context's target. The four members of
   this group should be considered as two pairs; the output from
   GSS_Sign() is properly input to GSS_Verify(), and the output from
   GSS_Seal() is properly input to GSS_Unseal().

   GSS_Sign() and GSS_Verify() support data origin authentication and
   data integrity services. When GSS_Sign() is invoked on an input
   message, it yields a per-message token containing data items which
   allow underlying mechanisms to provide the specified security
   services. The original message, along with the generated per-message
   token, is passed to the remote peer; these two data elements are
   processed by GSS_Verify(), which validates the message in
   conjunction with the separate token.

   GSS_Seal() and GSS_Unseal() support caller-requested confidentiality
   in addition to the data origin authentication and data integrity
   services offered by GSS_Sign() and GSS_Verify(). GSS_Seal() outputs
   a single data element, encapsulating optionally enciphered user data
   as well as associated token data items. The data element output from
   GSS_Seal() is passed to the remote peer and processed by
   GSS_Unseal() at that system. GSS_Unseal() combines decipherment (as
   required) with validation of data items related to authentication and
   integrity.

2.3.1. GSS_Sign call

   Inputs:

   o context_handle INTEGER,

   o qop_req INTEGER,-0 specifies default QOP

   o message OCTET STRING

   Outputs:

   o major_status INTEGER,

   o minor_status INTEGER,

   o per_msg_token OCTET STRING

   Return major_status codes:

   o GSS_COMPLETE indicates that a signature, suitable for an
      established security context, was successfully applied and that
      the message and corresponding per_msg_token are ready for
      transmission.

   o GSS_CONTEXT_EXPIRED indicates that context-related data items have
      expired, so that the requested operation cannot be performed.

   o GSS_CREDENTIALS_EXPIRED indicates that the context is recognized,
      but that its associated credentials have expired, so that the
      requested operation cannot be performed.

   o GSS_NO_CONTEXT indicates that no valid context was recognized for
      the input context_handle provided.

   o GSS_FAILURE indicates that the context is recognized, but that the
      requested operation could not be performed for reasons unspecified
      at the GSS-API level.

   Using the security context referenced by context_handle, apply a
   signature to the input message (along with timestamps and/or other
   data included in support of mech_type-specific mechanisms) and return
   the result in per_msg_token. The qop_req parameter allows quality-
   of-protection control. The caller passes the message and the
   per_msg_token to the target.

   The GSS_Sign() function completes before the message and
   per_msg_token is sent to the peer; successful application of
   GSS_Sign() does not guarantee that a corresponding GSS_Verify() has
   been (or can necessarily be) performed successfully when the message
   arrives at the destination.

2.3.2. GSS_Verify call

   Inputs:

   o context_handle INTEGER,

   o message OCTET STRING,

   o per_msg_token OCTET STRING

Outputs:

o   qop_state INTEGER,

o   major_status INTEGER,

o   minor_status INTEGER,

Return major_status codes:

o   GSS_COMPLETE indicates that the message was successfully verified.

o   GSS_DEFECTIVE_TOKEN indicates that consistency checks performed on
    the received per_msg_token failed, preventing further processing
    from being performed with that token.

o   GSS_BAD_SIG indicates that the received per_msg_token contains an
    incorrect signature for the message.

o   GSS_DUPLICATE_TOKEN, GSS_OLD_TOKEN, and GSS_UNSEQ_TOKEN values
    appear in conjunction with the optional per-message replay
    detection features described in Section 1.2.3; their semantics are
    described in that section.

o   GSS_CONTEXT_EXPIRED indicates that context-related data items have
    expired, so that the requested operation cannot be performed.

o   GSS_CREDENTIALS_EXPIRED indicates that the context is recognized,
    but that its associated credentials have expired, so that the
    requested operation cannot be performed.

o   GSS_NO_CONTEXT indicates that no valid context was recognized for
    the input context_handle provided.

o   GSS_FAILURE indicates that the context is recognized, but that the
    GSS_Verify() operation could not be performed for reasons
    unspecified at the GSS-API level.

Using the security context referenced by context_handle, verify that
the input per_msg_token contains an appropriate signature for the
input message, and apply any active replay detection or sequencing
features. Return an indication of the quality-of-protection applied
to the processed message in the qop_state result.

2.3.3. GSS_Seal call

   Inputs:

   o context_handle INTEGER,

   o conf_req_flag BOOLEAN,

   o qop_req INTEGER,-0 specifies default QOP

   o input_message OCTET STRING

   Outputs:

   o major_status INTEGER,

   o minor_status INTEGER,

   o conf_state BOOLEAN,

   o output_message OCTET STRING

   Return major_status codes:

   o GSS_COMPLETE indicates that the input_message was successfully
      processed and that the output_message is ready for transmission.

   o GSS_CONTEXT_EXPIRED indicates that context-related data items have
      expired, so that the requested operation cannot be performed.

   o GSS_CREDENTIALS_EXPIRED indicates that the context is recognized,
      but that its associated credentials have expired, so that the
      requested operation cannot be performed.

   o GSS_NO_CONTEXT indicates that no valid context was recognized for
      the input context_handle provided.

   o GSS_FAILURE indicates that the context is recognized, but that the
      GSS_Seal() operation could not be performed for reasons
      unspecified at the GSS-API level.

   Performs the data origin authentication and data integrity functions
   of GSS_Sign(). If the input conf_req_flag is TRUE, requests that
   confidentiality be applied to the input_message. Confidentiality may
   not be supported in all mech_types or by all implementations; the
   returned conf_state flag indicates whether confidentiality was
   provided for the input_message. The qop_req parameter allows
   quality-of-protection control.

In all cases, the GSS_Seal() call yields a single output_message
data element containing (optionally enciphered) user data as well as
control information.

2.3.4. GSS_Unseal call

Inputs:

o context_handle INTEGER,

o input_message OCTET STRING

Outputs:

o conf_state BOOLEAN,

o qop_state INTEGER,

o major_status INTEGER,

o minor_status INTEGER,

o output_message OCTET STRING

Return major_status codes:

o GSS_COMPLETE indicates that the input_message was successfully
   processed and that the resulting output_message is available.

o GSS_DEFECTIVE_TOKEN indicates that consistency checks performed on
   the per_msg_token extracted from the input_message failed,
   preventing further processing from being performed.

o GSS_BAD_SIG indicates that an incorrect signature was detected for
   the message.

o GSS_DUPLICATE_TOKEN, GSS_OLD_TOKEN, and GSS_UNSEQ_TOKEN values
   appear in conjunction with the optional per-message replay
   detection features described in Section 1.2.3; their semantics are
   described in that section.

o GSS_CONTEXT_EXPIRED indicates that context-related data items have
   expired, so that the requested operation cannot be performed.

o GSS_CREDENTIALS_EXPIRED indicates that the context is recognized,
   but that its associated credentials have expired, so that the
   requested operation cannot be performed.

o GSS_NO_CONTEXT indicates that no valid context was recognized for
   the input context_handle provided.

o GSS_FAILURE indicates that the context is recognized, but that the
   GSS_Unseal() operation could not be performed for reasons
   unspecified at the GSS-API level.

Processes a data element generated (and optionally enciphered) by
GSS_Seal(), provided as input_message. The returned conf_state value
indicates whether confidentiality was applied to the input_message.
If conf_state is TRUE, GSS_Unseal() deciphers the input_message.
Returns an indication of the quality-of-protection applied to the
processed message in the qop_state result. GSS_Seal() performs the
data integrity and data origin authentication checking functions of
GSS_Verify() on the plaintext data. Plaintext data is returned in
output_message.

2.4. Support calls

This group of calls provides support functions useful to GSS-API
callers, independent of the state of established contexts. Their
characterization with regard to blocking or non-blocking status in
terms of network interactions is unspecified.

2.4.1. GSS_Display_status call

Inputs:

o status_value INTEGER,-GSS-API major_status or minor_status
   return value

o status_type INTEGER,-1 if major_status, 2 if minor_status

o mech_type OBJECT IDENTIFIER-mech_type to be used for minor_
   status translation

Outputs:

o major_status INTEGER,

o minor_status INTEGER,

o status_string_set SET OF OCTET STRING

Return major_status codes:

o GSS_COMPLETE indicates that a valid printable status
   representation (possibly representing more than one status event

encoded within the status_value) is available in the returned
status_string_set.

o GSS_BAD_MECH indicates that translation in accordance with an
   unsupported mech_type was requested, so translation could not be
   performed.

o GSS_BAD_STATUS indicates that the input status_value was invalid,
   or that the input status_type carried a value other than 1 or 2,
   so translation could not be performed.

o GSS_FAILURE indicates that the requested operation could not be
   performed for reasons unspecified at the GSS-API level.

Provides a means for callers to translate GSS-API-returned major and
minor status codes into printable string representations.

2.4.2. GSS_Indicate_mechs call

Input:

o (none)

Outputs:

o major_status INTEGER,

o minor_status INTEGER,

o mech_set SET OF OBJECT IDENTIFIER

Return major_status codes:

o GSS_COMPLETE indicates that a set of available mechanisms has
   been returned in mech_set.

o GSS_FAILURE indicates that the requested operation could not
   be performed for reasons unspecified at the GSS-API level.

Allows callers to determine the set of mechanism types available on
the local system. This call is intended for support of specialized
callers who need to request non-default mech_type sets from
GSS_Acquire_cred(), and should not be needed by other callers.

2.4.3. GSS_Compare_name call

Inputs:

o name1 INTERNAL NAME,

o name2 INTERNAL NAME

Outputs:

o major_status INTEGER,

o minor_status INTEGER,

o name_equal BOOLEAN

Return major_status codes:

o GSS_COMPLETE indicates that name1 and name2 were comparable, and
   that the name_equal result indicates whether name1 and name2 were
   equal or unequal.

o GSS_BAD_NAMETYPE indicates that one or both of name1 and name2
   contained internal type specifiers uninterpretable by the
   supporting GSS-API implementation, or that the two names' types
   are different and incomparable, so the equality comparison could
   not be completed.

o GSS_BAD_NAME indicates that one or both of the input names was
   ill-formed in terms of its internal type specifier, so the
   equality comparison could not be completed.

o GSS_FAILURE indicates that the requested operation could not be
   performed for reasons unspecified at the GSS-API level.

Allows callers to compare two internal name representations for
equality.

2.4.4. GSS_Display_name call

   Inputs:

o name INTERNAL NAME

   Outputs:

o major_status INTEGER,

o minor_status INTEGER,

o name_string OCTET STRING,

o name_type OBJECT IDENTIFIER

Return major_status codes:

o GSS_COMPLETE indicates that a valid printable name representation
   is available in the returned name_string.

o GSS_BAD_NAMETYPE indicates that the provided name was of a type
   uninterpretable by the supporting GSS-API implementation, so no
   printable representation could be generated.

o GSS_BAD_NAME indicates that the contents of the provided name were
   inconsistent with the internally-indicated name type, so no
   printable representation could be generated.

o GSS_FAILURE indicates that the requested operation could not be
   performed for reasons unspecified at the GSS-API level.

Allows callers to translate an internal name representation into a
printable form with associated namespace type descriptor. The syntax
of the printable form is a local matter.

2.4.5. GSS_Import_name call

Inputs:

o input_name_string OCTET STRING,

o input_name_type OBJECT IDENTIFIER

Outputs:

o major_status INTEGER,

o minor_status INTEGER,

o output_name INTERNAL NAME

Return major_status codes:

o GSS_COMPLETE indicates that a valid name representation is output
   in output_name and described by the type value in
   output_name_type.

o GSS_BAD_NAMETYPE indicates that the input_name_type is unsupported
   by the GSS-API implementation, so the import operation could not
   be completed.

   o GSS_BAD_NAME indicates that the provided input_name_string is
      ill-formed in terms of the input_name_type, so the import
      operation could not be completed.

   o GSS_FAILURE indicates that the requested operation could not be
      performed for reasons unspecified at the GSS-API level.

   Allows callers to provide a printable name representation, designate
   the type of namespace in conjunction with which it should be parsed,
   and convert that printable representation to an internal form
   suitable for input to other GSS-API routines. The syntax of the
   input_name is a local matter.

2.4.6. GSS_Release_name call

   Inputs:

   o name INTERNAL NAME

   Outputs:

   o major_status INTEGER,

   o minor_status INTEGER

   Return major_status codes:

   o GSS_COMPLETE indicates that the storage associated with the input
      name was successfully released.

   o GSS_BAD_NAME indicates that the input name argument did not
      contain a valid name.

   o GSS_FAILURE indicates that the requested operation could not be
      performed for reasons unspecified at the GSS-API level.

   Allows callers to release the storage associated with an internal
   name representation.

2.4.7. GSS_Release_buffer call

   Inputs:

   o buffer OCTET STRING

   Outputs:

   o major_status INTEGER,

o   minor_status INTEGER

Return major_status codes:

o   GSS_COMPLETE indicates that the storage associated with the input
    buffer was successfully released.

o   GSS_FAILURE indicates that the requested operation could not be
    performed for reasons unspecified at the GSS-API level.

Allows callers to release the storage associated with an OCTET STRING
buffer allocated by another GSS-API call.

2.4.8. GSS_Release_oid_set call

Inputs:

o   buffer SET OF OBJECT IDENTIFIER

Outputs:

o   major_status INTEGER,

o   minor_status INTEGER

Return major_status codes:

o   GSS_COMPLETE indicates that the storage associated with the input
    object identifier set was successfully released.

o   GSS_FAILURE indicates that the requested operation could not be
    performed for reasons unspecified at the GSS-API level.

Allows callers to release the storage associated with an object
identifier set object allocated by another GSS-API call.

3. Mechanism-Specific Example Scenarios

This section provides illustrative overviews of the use of various
candidate mechanism types to support the GSS-API. These discussions
are intended primarily for readers familiar with specific security
technologies, demonstrating how GSS-API functions can be used and
implemented by candidate underlying mechanisms. They should not be
regarded as constrictive to implementations or as defining the only
means through which GSS-API functions can be realized with a
particular underlying technology, and do not demonstrate all GSS-API
features with each technology.

3.1. Kerberos V5, single-TGT

   OS-specific login functions yield a TGT to the local realm Kerberos
   server; TGT is placed in a credentials structure for the client.
   Client calls GSS_Acquire_cred() to acquire a cred_handle in order to
   reference the credentials for use in establishing security contexts.

   Client calls GSS_Init_sec_context(). If the requested service is
   located in a different realm, GSS_Init_sec_context() gets the
   necessary TGT/key pairs needed to traverse the path from local to
   target realm; these data are placed in the owner's TGT cache. After
   any needed remote realm resolution, GSS_Init_sec_context() yields a
   service ticket to the requested service with a corresponding session
   key; these data are stored in conjunction with the context. GSS-API
   code sends KRB_TGS_REQ request(s) and receives KRB_TGS_REP
   response(s) (in the successful case) or KRB_ERROR.

   Assuming success, GSS_Init_sec_context() builds a Kerberos-formatted
   KRB_AP_REQ message, and returns it in output_token. The client sends
   the output_token to the service.

   The service passes the received token as the input_token argument to
   GSS_Accept_sec_context(), which verifies the authenticator, provides
   the service with the client's authenticated name, and returns an
   output_context_handle.

   Both parties now hold the session key associated with the service
   ticket, and can use this key in subsequent GSS_Sign(), GSS_Verify(),
   GSS_Seal(), and GSS_Unseal() operations.

3.2. Kerberos V5, double-TGT

   TGT acquisition as above.

   Note: To avoid unnecessary frequent invocations of error paths when
   implementing the GSS-API atop Kerberos V5, it seems appropriate to
   represent "single-TGT K-V5" and "double-TGT K-V5" with separate
   mech_types, and this discussion makes that assumption.

   Based on the (specified or defaulted) mech_type,
   GSS_Init_sec_context() determines that the double-TGT protocol
   should be employed for the specified target. GSS_Init_sec_context()
   returns GSS_CONTINUE_NEEDED major_status, and its returned
   output_token contains a request to the service for the service's TGT.
   (If a service TGT with suitably long remaining lifetime already
   exists in a cache, it may be usable, obviating the need for this
   step.) The client passes the output_token to the service. Note: this
   scenario illustrates a different use for the GSS_CONTINUE_NEEDED

status return facility than for support of mutual authentication;
note that both uses can coexist as successive operations within a
single context establishment operation.

The service passes the received token as the input_token argument to
GSS_Accept_sec_context(), which recognizes it as a request for TGT.
(Note that current Kerberos V5 defines no intra-protocol mechanism to
represent such a request.) GSS_Accept_sec_context() returns
GSS_CONTINUE_NEEDED major_status and provides the service's TGT in
its output_token. The service sends the output_token to the client.

The client passes the received token as the input_token argument to a
continuation of GSS_Init_sec_context(). GSS_Init_sec_context() caches
the received service TGT and uses it as part of a service ticket
request to the Kerberos authentication server, storing the returned
service ticket and session key in conjunction with the context.
GSS_Init_sec_context() builds a Kerberos-formatted authenticator,
and returns it in output_token along with GSS_COMPLETE return
major_status. The client sends the output_token to the service.

Service passes the received token as the input_token argument to a
continuation call to GSS_Accept_sec_context().
GSS_Accept_sec_context() verifies the authenticator, provides the
service with the client's authenticated name, and returns
major_status GSS_COMPLETE.

GSS_Sign(), GSS_Verify(), GSS_Seal(), and GSS_Unseal() as above.

3.3. X.509 Authentication Framework

This example illustrates use of the GSS-API in conjunction with
public-key mechanisms, consistent with the X.509 Directory
Authentication Framework.

The GSS_Acquire_cred() call establishes a credentials structure,
making the client's private key accessible for use on behalf of the
client.

The client calls GSS_Init_sec_context(), which interrogates the
Directory to acquire (and validate) a chain of public-key
certificates, thereby collecting the public key of the service. The
certificate validation operation determines that suitable signatures
were applied by trusted authorities and that those certificates have
not expired. GSS_Init_sec_context() generates a secret key for use
in per-message protection operations on the context, and enciphers
that secret key under the service's public key.

The enciphered secret key, along with an authenticator quantity

signed with the client's private key, is included in the output_token
from GSS_Init_sec_context(). The output_token also carries a
certification path, consisting of a certificate chain leading from
the service to the client; a variant approach would defer this path
resolution to be performed by the service instead of being asserted
by the client. The client application sends the output_token to the
service.

The service passes the received token as the input_token argument to
GSS_Accept_sec_context(). GSS_Accept_sec_context() validates the
certification path, and as a result determines a certified binding
between the client's distinguished name and the client's public key.
Given that public key, GSS_Accept_sec_context() can process the
input_token's authenticator quantity and verify that the client's
private key was used to sign the input_token. At this point, the
client is authenticated to the service. The service uses its private
key to decipher the enciphered secret key provided to it for per-
message protection operations on the context.

The client calls GSS_Sign() or GSS_Seal() on a data message, which
causes per-message authentication, integrity, and (optional)
confidentiality facilities to be applied to that message. The service
uses the context's shared secret key to perform corresponding
GSS_Verify() and GSS_Unseal() calls.

4. Related Activities

In order to implement the GSS-API atop existing, emerging, and future
security mechanisms:

    object identifiers must be assigned to candidate GSS-API
    mechanisms and the name types which they support

    concrete data element formats must be defined for candidate
    mechanisms

Calling applications must implement formatting conventions which will
enable them to distinguish GSS-API tokens from other data carried in
their application protocols.

Concrete language bindings are required for the programming
environments in which the GSS-API is to be employed; such bindings
for the C language are available in an associated RFC.

5. Acknowledgments

   This proposal is the result of a collaborative effort.
   Acknowledgments are due to the many members of the IETF Security Area
   Advisory Group (SAAG) and the Common Authentication Technology (CAT)
   Working Group for their contributions at meetings and by electronic
   mail. Acknowledgments are also due to Kannan Alagappan, Doug Barlow,
   Bill Brown, Cliff Kahn, Charlie Kaufman, Butler Lampson, Richard
   Pitkin, Joe Tardo, and John Wray of Digital Equipment Corporation,
   and John Carr, John Kohl, Jon Rochlis, Jeff Schiller, and Ted T'so of
   MIT and Project Athena. Joe Pato and Bill Sommerfeld of HP/Apollo,
   Walt Tuvell of OSF, and Bill Griffith and Mike Merritt of AT&T,
   provided inputs which helped to focus and clarify directions.
   Precursor work by Richard Pitkin, presented to meetings of the
   Trusted Systems Interoperability Group (TSIG), helped to demonstrate
   the value of a generic, mechanism-independent security service API.

6. Security Considerations

   Security issues are discussed throughout this memo.

7. Author's Address

   John Linn
   Geer Zolot Associates
   One Main St.
   Cambridge, MA 02142 USA

   Phone: +1 617.374.3700
   Email: Linn@gza.com

APPENDIX A

PACS AND AUTHORIZATION SERVICES

   Consideration has been given to modifying the GSS-API service
   interface to recognize and manipulate Privilege Attribute
   Certificates (PACs) as in ECMA 138, carrying authorization data as a
   side effect of establishing a security context, but no such
   modifications have been incorporated at this time. This appendix
   provides rationale for this decision and discusses compatibility
   alternatives between PACs and the GSS-API which do not require that
   PACs be made visible to GSS-API callers.

   Existing candidate mechanism types such as Kerberos and X.509 do not
   incorporate PAC manipulation features, and exclusion of such
   mechanisms from the set of candidates equipped to fully support the
   GSS-API seems inappropriate. Inclusion (and GSS-API visibility) of a
   feature supported by only a limited number of mechanisms could
   encourage the development of ostensibly portable applications which
   would in fact have only limited portability.

   The status quo, in which PACs are not visible across the GSS-API
   interface, does not preclude implementations in which PACs are
   carried transparently, within the tokens defined and used for certain
   mech_types, and stored within peers' credentials and context-level
   data structures. While invisible to API callers, such PACs could be
   used by operating system or other local functions as inputs in the
   course of mediating access requests made by callers. This course of
   action allows dynamic selection of PAC contents, if such selection is
   administratively-directed rather than caller-directed.

   In a distributed computing environment, authentication must span
   different systems; the need for such authentication provides
   motivation for GSS-API definition and usage. Heterogeneous systems in
   a network can intercommunicate, with globally authenticated names
   comprising the common bond between locally defined access control
   policies. Access control policies to which authentication provides
   inputs are often local, or specific to particular operating systems
   or environments. If the GSS-API made particular authorization models
   visible across its service interface, its scope of application would
   become less general. The current GSS-API paradigm is consistent with
   the precedent set by Kerberos, neither defining the interpretation of
   authorization-related data nor enforcing access controls based on
   such data.

   The GSS-API is a general interface, whose callers may reside inside
   or outside any defined TCB or NTCB boundaries. Given this
   characteristic, it appears more realistic to provide facilities which

provide "value-added" security services to its callers than to offer
facilities which enforce restrictions on those callers. Authorization
decisions must often be mediated below the GSS-API level in a local
manner against (or in spite of) applications, and cannot be
selectively invoked or omitted at those applications' discretion.
Given that the GSS-API's placement prevents it from providing a
comprehensive solution to the authorization issue, the value of a
partial contribution specific to particular authorization models is
debatable.

APPENDIX B

MECHANISM-INDEPENDENT TOKEN FORMAT

This appendix specifies a mechanism-independent level of
encapsulating representation for the initial token of a GSS-API
context establishment sequence, incorporating an identifier of the
mechanism type to be used on that context. Use of this format (with
ASN.1-encoded data elements represented in BER, constrained in the
interests of parsing simplicity to the Distinguished Encoding Rule
(DER) BER subset defined in X.509, clause 8.7) is recommended to the
designers of GSS-API implementations based on various mechanisms, so
that tokens can be interpreted unambiguously at GSS-API peers. There
is no requirement that the mechanism-specific innerContextToken,
innerMsgToken, and sealedUserData data elements be encoded in ASN.1
BER.

```
     -- optional top-level token definitions to
     -- frame different mechanisms

     GSS-API DEFINITIONS ::=

     BEGIN

     MechType ::= OBJECT IDENTIFIER
     -- data structure definitions

     -- callers must be able to distinguish among
     -- InitialContextToken, SubsequentContextToken,
     -- PerMsgToken, and SealedMessage data elements
     -- based on the usage in which they occur

     InitialContextToken ::=
     -- option indication (delegation, etc.) indicated within
     -- mechanism-specific token
     [APPLICATION 0] IMPLICIT SEQUENCE {
             thisMech MechType,
             innerContextToken ANY DEFINED BY thisMech
```

```
                     -- contents mechanism-specific
                 }

          SubsequentContextToken ::= innerContextToken ANY
          -- interpretation based on predecessor InitialContextToken

          PerMsgToken ::=
          -- as emitted by GSS_Sign and processed by GSS_Verify
                  innerMsgToken ANY

          SealedMessage ::=
          -- as emitted by GSS_Seal and processed by GSS_Unseal
          -- includes internal, mechanism-defined indicator
          -- of whether or not encrypted
                  sealedUserData ANY

          END
```

APPENDIX C

MECHANISM DESIGN CONSTRAINTS

   The following constraints on GSS-API mechanism designs are adopted in
   response to observed caller protocol requirements, and adherence
   thereto is anticipated in subsequent descriptions of GSS-API
   mechanisms to be documented in standards-track Internet
   specifications.

   Use of the approach defined in Appendix B of this specification,
   applying a mechanism type tag to the InitialContextToken, is
   required.

   It is strongly recommended that mechanisms offering per-message
   protection services also offer at least one of the replay detection
   and sequencing services, as mechanisms offering neither of the latter
   will fail to satisfy recognized requirements of certain candidate
   caller protocols.