Internet-Draft   Jack Kabat
IETF CAT Working Group   ValiCert, Inc.
Document: <draft-ietf-cat-gssv2-javabind-03.txt>   Mayank Upadhyay
                                                  Sun Microsystems, Inc.

                                                        October 1999

Generic Security Service API Version 2 : Java bindings

Status of this Memo

Abstract

   The Generic Security Services Application Program Interface (GSS-API)
   offers application programmers uniform access to security services
   atop a variety of underlying cryptographic mechanisms. This document
   specifies the Java bindings for GSS-API which is described at a
   language independent conceptual level in RFC 2078 [GSSAPIv2].

   The GSS-API allows a caller application to authenticate a principal
   identity, to delegate rights to a peer, and to apply security
   services such as confidentiality and integrity on a per-message
   basis. Examples of security mechanisms defined for GSS-API are The
   Simple Public-Key GSS-API Mechanism [SPKM] and The Kerberos Version 5
   GSS-API Mechanism [KERBV5].

Table of Contents

[1]. **Introduction**

   This document specifies Java language bindings for the Generic
   Security Services Application Programming Interface (GSS-API) Version
   2. GSS-API Version 2 is described in a language independent format in
   RFC 2078 [GSSAPIv2]. The GSS-API allows a caller application to
   authenticate a principal identity, to delegate rights to a peer, and
   to apply security services such as confidentiality and integrity on a
   per-message basis.

   This document leverages the work performed by the WG in the area of
   RFC 2078 [GSSAPIv2] the C-bindings draft [GSSAPI-C]. Whenever
   appropriate, text has been used from the C-bindings document to
   explain generic concepts and provide direction to the implementors.

   The design goals of this API have been to satisfy all the
   functionality defined in RFC 2078 and to provide these services in an
   object oriented method. The specification also aims to satisfy the
   needs of both types of Java application developers, those who would
   like access to a "system-wide" GSS-API implementation, as well as
   those who would want to provide their own "custom" implementation.

   A "system-wide" implementation is one that is available to all
   applications in the form of a library package. It may be a standard
   package in the Java runtime environment (JRE) being used or it may be
   additionally installed and accessible to any application via the
   CLASSPATH.

   A "custom" implementation of the GSS-API, on the other hand, is one
   that would, in most cases, be bundled with the application during
   distribution. It is expected that such an implementation would be
   meant to provide for some particular need of the application, such as
   support for some specific mechanism.

   The design of this API also aims to provide a flexible framework to
   add and manage GSS-API mechanisms. GSS-API leverages the JCA provider
   concept to support the plugability of mechanisms. Mechanisms can be
   added on a "system-wide" basis, where all users of the framework will
   have them available. The specification also allows for the addition
   of mechanisms per-instance the GSS-API.

   Lastly, this specification presents an API that will naturally fit
   within the operation environment of the Java platform. Readers are
   assumed to be familiar with both the GSS-API and the Java platform.

[2](#). **GSS-API Operational Paradigm**

   The Generic Security Service Application Programming Interface
   [[GSSAPIv2](#)] defines a generic security API to calling applications.
   It allows a communicating application to authenticate the user
   associated with another application, to delegate rights to another
   application, and to apply security services such as confidentiality
   and integrity on a per-message basis.

      There are four stages to using GSS-API:

      1)   The application acquires a set of credentials with which it
           may prove its identity to other processes. The
           application's credentials vouch for its global identity,
           which may or may not be related to any local username under
           which it may be running.

      2)   A pair of communicating applications establish a joint
           security context using their credentials. The security
           context encapsulates shared state information, which is
           required in order that per-message security services may be
           provided. Examples of state information that might be
           shared between applications as part of a security context
           are cryptographic keys, and message sequence numbers. As
           part of the establishment of a security context, the
           context initiator is authenticated to the responder, and
           may require that the responder is authenticated back to the
           initiator. The initiator may optionally give the responder
           the right to initiate further security contexts, acting as
           an agent or delegate of the initiator. This transfer of
           rights is termed "delegation", and is achieved by creating
           a set of credentials, similar to those used by the
           initiating application, but which may be used by the
           responder.

           A GSSContext object is used to establish and maintain the
           shared information that makes up the security context.
           Certain GSSContext methods will generate a token, which
           applications treat as cryptographically protected, opaque
           data. The caller of such GSSContext method is responsible
           for transferring the token to the peer application,
           encapsulated if necessary in an application-to-application
           protocol. On receipt of such a token, the peer application
           should pass it to a corresponding GSSContext method which
           will decode the token and extract the information, updating
           the security context state information accordingly.

3)   Per-message services are invoked on a GSSContext object to
      apply either:

          integrity and data origin authentication, or

          confidentiality, integrity and data origin
          authentication

      to application data, which are treated by GSS-API as
      arbitrary octet-strings. An application transmitting a
      message that it wishes to protect will call the appropriate
      GSSContext method (getMIC or wrap) to apply protection, and
      send the resulting token to the receiving application. The
      receiver will pass the received token (and, in the case of
      data protected by getMIC, the accompanying message-data) to
      the corresponding decoding method of the GSSContext
      interface (verifyMIC or unwrap) to remove the protection
      and validate the data.

4)   At the completion of a communications session (which may
      extend across several transport connections), each
      application uses a GSSContext method to invalidate the
      security context and release any system or cryptographic
      resources held. Multiple contexts may also be used (either
      successively or simultaneously) within a single
      communications association, at the discretion of the
      applications.

3. Additional Controls

This section discusses the optional services that a context initiator
may request of the GSS-API before the context establishment. Each of
these services is requested by calling the appropriate mutator method
in the GSSContext object before the first call to init is performed.
Only the context initiator can request context flags.

The optional services defined are:

Delegation
      The (usually temporary) transfer of rights from initiator to
      acceptor, enabling the acceptor to authenticate itself as an
      agent of the initiator.

Mutual Authentication
      In addition to the initiator authenticating its identity to the

     context acceptor, the context acceptor should also authenticate
     itself to the initiator.

  Replay Detection
     In addition to providing message integrity services, GSSContext
     per-message operations of getMIC and wrap should include message
     numbering information to enable verifyMIC and unwrap to detect
     if a message has been duplicated.

  Out-of-Sequence Detection
     In addition to providing message integrity services, GSSContext
     per-message operations (getMIC and wrap) should include message
     sequencing information to enable verifyMIC and unwrap to detect
     if a message has been received out of sequence.

  Anonymous Authentication
     The establishment of the security context should not reveal the
     initiator's identity to the context acceptor.

  Some mechanisms may not support all optional services, and some
  mechanisms may only support some services in conjunction with others.
  The GSSContext interface offers query methods to allow the
  verification by the calling application of which services will be
  available from the context when the establishment phase is complete.
  In general, if the security mechanism is capable of providing a
  requested service, it should do so even if additional services must
  be enabled in order to provide the requested service. If the
  mechanism is incapable of providing a requested service, it should
  proceed without the service leaving the application to abort the
  context establishment process if it considers the requested service
  to be mandatory.

  Some mechanisms may specify that support for some services is
  optional, and that implementors of the mechanism need not provide it.
  This is most commonly true of the confidentiality service, often
  because of legal restrictions on the use of data-encryption, but may
  apply to any of the services. Such mechanisms are required to send
  at least one token from acceptor to initiator during context
  establishment when the initiator indicates a desire to use such a
  service, so that the initiating GSS-API can correctly indicate
  whether the service is supported by the acceptor's GSS-API.

3.1. Delegation

  The GSS-API allows delegation to be controlled by the initiating
  application via the requestCredDeleg method before the first call to
  init has been issued. Some mechanisms do not support delegation, and

   for such mechanisms attempts by an application to enable delegation
   are ignored.

   The acceptor of a security context, for which the initiator enabled
   delegation, can check if delegation was enabled by using the
   getCredDelegState method of the GSSContext interface. In cases when
   it is, the delegated credential object can be obtained by calling the
   getDelegCred method. The obtained GSSCredential object may then be
   used to initiate subsequent GSS-API security contexts as an agent or
   delegate of the initiator. If the original initiator's identity is
   "A" and the delegate's identity is "B", then, depending on the
   underlying mechanism, the identity embodied by the delegated
   credential may be either "A" or "B acting for A".

   For many mechanisms that support delegation, a simple boolean does
   not provide enough control. Examples of additional aspects of
   delegation control that a mechanism might provide to an application
   are duration of delegation, network addresses from which delegation
   is valid, and constraints on the tasks that may be performed by a
   delegate. Such controls are presently outside the scope of the GSS-
   API. GSS-API implementations supporting mechanisms offering
   additional controls should provide extension routines that allow
   these controls to be exercised (perhaps by modifying the initiator's
   GSS-API credential object prior to its use in establishing a
   context). However, the simple delegation control provided by GSS-API
   should always be able to over-ride other mechanism-specific
   delegation controls. If the application instructs the GSSContext
   object that delegation is not desired, then the implementation must
   not permit delegation to occur. This is an exception to the general
   rule that a mechanism may enable services even if they are not
   requested - delegation may only be provided at the explicit request
   of the application.

3.2. Mutual Authentication

   Usually, a context acceptor will require that a context initiator
   authenticate itself so that the acceptor may make an access-control
   decision prior to performing a service for the initiator. In some
   cases, the initiator may also request that the acceptor authenticate
   itself. GSS-API allows the initiating application to request this
   mutual authentication service by calling the requestMutualAuth method
   of the GSSContext interface with a "true" parameter before making the
   first call to init. The initiating application is informed as to
   whether or not the context acceptor has authenticated itself. Note
   that some mechanisms may not support mutual authentication, and other
   mechanisms may always perform mutual authentication, whether or not
   the initiating application requests it. In particular, mutual

   authentication may be required by some mechanisms in order to support
   replay or out-of-sequence message detection, and for such mechanisms
   a request for either of these services will automatically enable
   mutual authentication.

3.3. **Replay and Out-of-Sequence Detection**

   The GSS-API may provide detection of mis-ordered messages once a
   security context has been established. Protection may be applied to
   messages by either application, by calling either getMIC or wrap
   methods of the GSSContext interface, and verified by the peer
   application by calling verifyMIC or unwrap for the peer's GSSContext
   object.

   The getMIC method calculates a cryptographic checksum of an
   application message, and returns that checksum in a token. The
   application should pass both the token and the message to the peer
   application, which presents them to the verifyMIC method of the
   peer's GSSContext object.

   The wrap method calculates a cryptographic checksum of an application
   message, and places both the checksum and the message inside a single
   token. The application should pass the token to the peer
   application, which presents it to the unwrap method of the peer's
   GSSContext object to extract the message and verify the checksum.

   Either pair of routines may be capable of detecting out-of-sequence
   message delivery, or duplication of messages. Details of such mis-
   ordered messages are indicated through supplementary query methods of
   the MessageProp object that is filled in by each of these routines.

   A mechanism need not maintain a list of all tokens that have been
   processed in order to support these status codes. A typical
   mechanism might retain information about only the most recent "N"
   tokens processed, allowing it to distinguish duplicates and missing
   tokens within the most recent "N" messages; the receipt of a token
   older than the most recent "N" would result in the isOldToken method
   of the instance of MessageProp to return "true".

3.4. **Anonymous Authentication**

   In certain situations, an application may wish to initiate the
   authentication process to authenticate a peer, without revealing its
   own identity. As an example, consider an application providing
   access to a database containing medical information, and offering
   unrestricted access to the service. A client of such a service might

   wish to authenticate the service (in order to establish trust in any
   information retrieved from it), but might not wish the service to be
   able to obtain the client's identity (perhaps due to privacy concerns
   about the specific inquiries, or perhaps simply to avoid being placed
   on mailing-lists).

   In normal use of the GSS-API, the initiator's identity is made
   available to the acceptor as a result of the context establishment
   process. However, context initiators may request that their identity
   not be revealed to the context acceptor. Many mechanisms do not
   support anonymous authentication, and for such mechanisms the request
   will not be honored. An authentication token will still be
   generated, but the application is always informed if a requested
   service is unavailable, and has the option to abort context
   establishment if anonymity is valued above the other security
   services that would require a context to be established.

   In addition to informing the application that a context is
   established anonymously (via the isAnonymous method of the GSSContext
   class), the getSrcName method of the acceptor's GSSContext object
   will, for such contexts, return a reserved internal-form name,
   defined by the implementation.

   The toString method for a GSSName object representing an anonymous
   entity will return a printable name. The returned value will be
   syntactically distinguishable from any valid principal name supported
   by the implementation. The associated name-type object identifier
   will be an oid representing the value of NT_ANONYMOUS. This name-
   type oid will be defined as a public, static Oid object of the
   GSSName class. The printable form of an anonymous name should be
   chosen such that it implies anonymity, since this name may appear in,
   for example, audit logs. For example, the string "<anonymous>" might
   be a good choice, if no valid printable names supported by the
   implementation can begin with "<" and end with ">".

   When using the equal method of the GSSName interface, and one of the
   operands is a GSSName instance representing an anonymous entity, the
   method must return "false".

3.5. Confidentiality

   If a GSSContext supports the confidentiality service, wrap method may
   be used to encrypt application messages. Messages are selectively
   encrypted, under the control of the setPrivacy method of the
   MessageProp object used in the wrap method.

3.6. **Inter-process Context Transfer**

   GSS-API V2 provides functionality which allows a security context to
   be transferred between processes on a single machine. These are
   implemented using the export method of GSSContext and a byte array
   constructor of the same class. The most common use for such a
   feature is a client-server design where the server is implemented as
   a single process that accepts incoming security contexts, which then
   launches child processes to deal with the data on these contexts. In
   such a design, the child processes must have access to the security
   context object created within the parent so that they can use per-
   message protection services and delete the security context when the
   communication session ends.

   Since the security context data structure is expected to contain
   sequencing information, it is impractical in general to share a
   context between processes. Thus GSSContext interface provides an
   export method that the process, which currently owns the context, can
   call to declare that it has no intention to use the context
   subsequently, and to create an inter-process token containing
   information needed by the adopting process to successfully re-create
   the context. After successful completion of export, the original
   security context is made inaccessible to the calling process by GSS-
   API and any further usage of this object will result in failures.
   The originating process transfers the inter-process token to the
   adopting process, which creates a new GSSContext object using the
   byte array constructor. The properties of the context are equivalent
   to that of the original context.

   The inter-process token may contain sensitive data from the original
   security context (including cryptographic keys). Applications using
   inter-process tokens to transfer security contexts must take
   appropriate steps to protect these tokens in transit.

   Implementations are not required to support the inter-process
   transfer of security contexts. Calling the isTransferable method of
   the GSSContext interface will indicate if the context object is
   transferable.

3.7. **The Use of Incomplete Contexts**

   Some mechanisms may allow the per-message services to be used before
   the context establishment process is complete. For example, a
   mechanism may include sufficient information in its initial context-
   level tokens for the context acceptor to immediately decode messages
   protected with wrap or getMIC. For such a mechanism, the initiating
   application need not wait until subsequent context-level tokens have

been sent and received before invoking the per-message protection
services.

An application can invoke the isProtReady method of the GSSContext
class to determine if the per-message services are available in
advance of complete context establishment. Applications wishing to
use per-message protection services on partially-established contexts
should query this method before attempting to invoke wrap or getMIC.

4. Calling Conventions

Java provides the implementors with not just a syntax for the
language, but also an operational environment. For example, memory
is automatically managed and does not require application
intervention. These language features have allowed for a simpler API
and have led to the elimination of certain GSS-API functions.

Moreover, the Java Cryptographic Architecture (JCA) defines a
provider model which allows an implementation independent access to
its security services. Using this model, applications can seamlessly
switch between different implementations and dynamically add new
services. The GSS-API specification leverages these concepts by the
usage of providers for the mechanism implementations.

4.1. Package Name

The classes and interfaces defined in this document reside in the
package called "org.ietf.jgss". Applications that wish to make use
of this API should import this package name as shows in section 7.

4.2. Provider Framework

The Java security API's use a provider architecture that allows
applications to be implementation independent and security API
implementations to be modular and extensible. The
java.security.Provider class is an abstract class that a vendor
extends. This class maps various properties that represent different
security services that are available to the names of the actual
vendor classes that implement those services. When requesting a
service, an application simply specifies the desired provider and the
API delegates the request to service classes available from that
provider.

   Using the Java security provider model insulates applications from
   implementation details of the services they wish to use.
   Applications can switch between providers easily and new providers
   can be added as needed, even at runtime.

   The GSS-API may use providers to find components for specific
   underlying security mechanisms. For instance, a particular provider
   might contain components that will allow the GSS-API to support the
   Kerberos v5 mechanism and another might contain components to support
   the SPKM mechanism. By delegating mechanism specific functionality
   to the components obtained from providers the GSS-API can be extended
   to support an arbitrary list of mechanism.

   How the GSS-API locates and queries these providers is beyond the
   scope of this document and is being deferred to a Service Provider
   Interface (SPI) specification. The availability of such a SPI
   specification is not mandatory for the adoption of this API
   specification nor is it mandatory to use providers in the
   implementation of a GSS-API framework. However, by using the provider
   framework together with an SPI specification one can create an
   extensible and implementation independent GSS-API framework.

4.3. Integer types

   All numeric values are declared as "int" primitive Java type. The
   Java specification guarantees that this will be a 32 bit two's
   complement signed number.

   Throughout this API, the "boolean" primitive Java type is used
   wherever a boolean value is required or returned.

4.4. Opaque Data types

   Java byte arrays are used to represent opaque data types which are
   consumed and produced by the GSS-API in the forms of tokens. Java
   arrays contain a length field which enables the users to easily
   determine their size. The language has automatic garbage collection
   which alleviates the need by developers to release memory and
   simplifies buffer ownership issues.

4.5. Strings

   The String object will be used to represent all textual data. The
   Java String object, transparently treats all characters as two-byte
   Unicode characters which allows support for many locals. All

routines returning or accepting textual data will use the String
object.

4.6. **Object Identifiers**

An Oid object will be used to represent Universal Object Identifiers
(Oids). Oids are ISO-defined, hierarchically globally-interpretable
identifiers used within the GSS-API framework to identify security
mechanisms and name formats. The Oid object can be created from a
string representation of its dot notation (e.g. "1.3.6.1.5.6.2") as
well as from its ASN.1 DER encoding. Methods are also provided to
test equality and provide the DER representation for the object.

An important feature of the Oid class is that its instances are
immutable - i.e. there are no methods defined that allow one to
change the contents of an Oid. This property allows one to treat
these objects as "statics" without the need to perform copies.

Certain routines allow the usage of a default oid. A "null" value
can be used in those cases.

4.7. **Object Identifier Sets**

The Java bindings represents object identifiers sets as arrays of Oid
objects. All Java arrays contain a length field which allows for
easy manipulation and reference.

In order to support the full functionality of RFC 2078, the Oid class
includes a method which checks for existence of an Oid object within
a specified array. This is equivalent in functionality to
gss_test_oid_set_member. The use of Java arrays and Java's automatic
garbage collection has eliminated the need for the following
routines: gss_create_empty_oid_set, gss_release_oid_set, and
gss_add_oid_set_member. Java GSS-API implementations will not
contain them. Java's automatic garbage collection and the immutable
property of the Oid object eliminates the complicated memory
management issues of the C counterpart.

When ever a default value for an Object Identifier Set is required, a
"null" value can be used. Please consult the detailed method
description for details.

4.8. **Credentials**

GSS-API credentials are represented by the GSSCredential interface.

The interface contains several constructs to allow for the creation
of most common credential objects for the initiator and the acceptor.
Comparisons are performed using the interface's "equals" method. The
following general description of GSS-API credentials is included from
the C-bindings specification:

GSS-API credentials can contain mechanism-specific principal
authentication data for multiple mechanisms. A GSS-API credential is
composed of a set of credential-elements, each of which is applicable
to a single mechanism. A credential may contain at most one
credential-element for each supported mechanism. A credential-
element identifies the data needed by a single mechanism to
authenticate a single principal, and conceptually contains two
credential-references that describe the actual mechanism-specific
authentication data, one to be used by GSS-API for initiating
contexts, and one to be used for accepting contexts. For mechanisms
that do not distinguish between acceptor and initiator credentials,
both references would point to the same underlying mechanism-specific
authentication data.

Credentials describe a set of mechanism-specific principals, and give
their holder the ability to act as any of those principals. All
principal identities asserted by a single GSS-API credential should
belong to the same entity, although enforcement of this property is
an implementation-specific matter. A single GSSCredential object
represents all the credential elements that have been acquired.

The creation's of an GSSContext object allows the value of "null" to
be specified as the GSSCredential input parameter. This will
indicate a desire by the application to act as a default principal.
While individual GSS-API implementations are free to determine such
default behavior as appropriate to the mechanism, the following
default behavior by these routines is recommended for portability:

     For the initiator side of the context:

     1)   If there is only a single principal capable of initiating
          security contexts for the chosen mechanism that the
          application is authorized to act on behalf of, then that
          principal shall be used, otherwise

     2)   If the platform maintains a concept of a default network-
          identity for the chosen mechanism, and if the application
          is authorized to act on behalf of that identity for the
          purpose of initiating security contexts, then the principal
          corresponding to that identity shall be used, otherwise

     3)   If the platform maintains a concept of a default local

identity, and provides a means to map local identities into
network-identities for the chosen mechanism, and if the
application is authorized to act on behalf of the network-
identity image of the default local identity for the
purpose of initiating security contexts using the chosen
mechanism, then the principal corresponding to that
identity shall be used, otherwise

4)   A user-configurable default identity should be used.

and for the acceptor side of the context

1)   If there is only a single authorized principal identity
capable of accepting security contexts for the chosen
mechanism, then that principal shall be used, otherwise

2)   If the mechanism can determine the identity of the target
principal by examining the context-establishment token
processed during the accept method, and if the accepting
application is authorized to act as that principal for the
purpose of accepting security contexts using the chosen
mechanism, then that principal identity shall be used,
otherwise

3)   If the mechanism supports context acceptance by any
principal, and if mutual authentication was not requested,
any principal that the application is authorized to accept
security contexts under using the chosen mechanism may be
used, otherwise

4)   A user-configurable default identity shall be used.

The purpose of the above rules is to allow security contexts to be
established by both initiator and acceptor using the default behavior
whenever possible. Applications requesting default behavior are
likely to be more portable across mechanisms and implementations than
ones that instantiate an GSSCredential object representing a specific
identity.

4.9. Contexts

The GSSContext interface is used to represent one end of a GSS-API
security context, storing state information appropriate to that end
of the peer communication, including cryptographic state information.

   The instantiation of the context object is done differently by the
   initiator and the acceptor. After the context has been instantiated,
   the initiator may choose to set various context options which will
   determine the characteristics of the desired security context. When
   all the application desired characteristics have been set, the
   initiator will call the initSecContext method which will produce a
   token for consumption by the peer's acceptSecContext method. It is
   the responsibility of the application to deliver the authentication
   token(s) between the peer applications for processing. Upon
   completion of the context establishment phase, context attributes can
   be retrieved, by both the initiator and acceptor, using the accessor
   methods. These will reflect the actual attributes of the established
   context. At this point the context can be used by the application to
   apply cryptographic services to its data.

4.10. Authentication tokens

   A token is a caller-opaque type that GSS-API uses to maintain
   synchronization between each end of the GSS-API security context.
   The token is a cryptographically protected octet-string, generated by
   the underlying mechanism at one end of a GSS-API security context for
   use by the peer mechanism at the other end. Encapsulation (if
   required) within the application protocol and transfer of the token
   are the responsibility of the peer applications.

   Java GSS-API uses byte arrays to represent authentication tokens.
   Overloaded methods exist which allow the caller to supply input and
   output streams which will be used for the reading and writing of the
   token data.

4.11. Interprocess tokens

   Certain GSS-API routines are intended to transfer data between
   processes in multi-process programs. These routines use a caller-
   opaque octet-string, generated by the GSS-API in one process for use
   by the GSS-API in another process. The calling application is
   responsible for transferring such tokens between processes. Note
   that, while GSS-API implementors are encouraged to avoid placing
   sensitive information within interprocess tokens, or to
   cryptographically protect them, many implementations will be unable
   to avoid placing key material or other sensitive data within them.
   It is the application's responsibility to ensure that interprocess
   tokens are protected in transit, and transferred only to processes
   that are trustworthy. An interprocess token is represented using a
   byte array emitted from the export method of the GSSContext
   interface. The receiver of the interprocess token would initialize

   an GSSContext object with this token to create a new context. Once a
   context has been exported, the GSSContext object is invalidated and
   is no longer available.

4.12. Error Reporting

   RFC 2078 defined the usage of major and minor status values for
   signaling of GSS-API errors. The major code, also called GSS status
   code, is used to signal errors at the GSS-API level independent of
   the underlying mechanism(s). The minor status value or Mechanism
   status code, is a mechanism defined error value indicating a
   mechanism specific error code.

   Java GSS-API uses exceptions implemented by the GSSException class to
   signal both minor and major error values. Both, mechanism specific
   errors and GSS-API level errors are signaled through instances of
   this class. The usage of exceptions replaces the need for major and
   minor codes to be used within the API calls. GSSException class also
   contains methods to obtain textual representations for both the major
   and minor values, which is equivalent to the functionality of
   gss_display_status.

4.12.1. GSS status codes

   GSS status codes indicate errors that are independent of the
   underlying mechanism(s) used to provide the security service. The
   errors that can be indicated via a GSS status code are generic API
   routine errors (errors that are defined in the GSS-API
   specification). These bindings take advantage of the Java exceptions
   mechanism, thus eliminating the need for calling errors.

   A GSS status code indicates a single fatal generic API error from the
   routine that has thrown the GSSException. Using exceptions announces
   that a fatal error has occurred during the execution of the method.
   The GSS-API operational model also allows for the signaling of
   supplementary status information from the per-message calls. These
   need to be handled as return values since using exceptions is not
   appropriate for informatory or warning-like information. The methods
   that are capable of producing supplementary information are the two
   per-message methods GSSContext.verifyMIC() and GSSContext.unwrap().
   These methods fill the supplementary status codes in the MessageProp
   object that was passed in.

   GSSException object, along with providing the functionality for
   setting of the various error codes and translating them into textual
   representation, also contains the definitions of all the numeric

error values. The following table lists the definitions of error
codes:

Table: GSS Status Codes

Name	Value	Meaning
BAD_MECH	1	An unsupported mechanism was requested.
BAD_NAME	2	An invalid name was supplied.
BAD_NAMETYPE	3	A supplied name was of an unsupported type.
BAD_BINDINGS	4	Incorrect channel bindings were supplied.
BAD_STATUS	5	An invalid status code was supplied.
BAD_MIC	6	A token had an invalid MIC.
NO_CRED	7	No credentials were supplied, or the credentials were unavailable or inaccessible.
NO_CONTEXT	8	Invalid context has been supplied.
DEFECTIVE_TOKEN	9	A supplied token was invalid.
DEFECTIVE_CREDENTIAL	10	A supplied credential was invalid.
CREDENTIALS_EXPIRED	11	The referenced credentials have expired.
CONTEXT_EXPIRED	12	The context has expired.
FAILURE	13	Miscellaneous failure, unspecified at the GSS-API level.
BAD_QOP	14	The quality-of-protection requested could not be provided.
UNAUTHORIZED	15	The operation is forbidden by

                              local security policy.

        UNAVAILABLE   16   The operation or option is
                                        unavailable.

        DUPLICATE_ELEMENT   17   The requested credential
                                        element already exists.

        NAME_NOT_MN   18   The provided name was not a
                                        mechanism name.

        OLD_TOKEN   19   The token's validity period has
                                        expired.

        DUPLICATE_TOKEN   20   The token was a duplicate of an
                                        earlier version.

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

4.12.2. Mechanism-specific status codes

    Mechanism-specific status codes are communicated in two ways, they
    are part of any GSSException thrown from the mechanism specific layer
    to signal a fatal error, or they are part of the MessageProp object
    that the per-message calls use to signal non-fatal errors.

    A default value of 0 in either the GSSException object or the
    MessageProp object will be used to represent the absence of any
    mechanism specific status code.

4.12.3. Supplementary status codes

    Supplementary status codes are confined to the per-message methods of
    the GSSContext interface. Because of the informative nature of these
    errors it is not appropriate to use exceptions to signal them.
    Instead, the per-message operations of the GSSContext interface
    return these values in a MessageProp object.

    The MessageProp class defines query methods which return boolean
    values indicating the following supplementary states:

      Table: Supplementary Status Methods

        Method Name   Meaning when "true" is returned

      isDuplicateToken   The token was a duplicate of an
                          earlier token.

      isOldToken   The token's validity period has
                          expired.

      isUnseqToken   A later token has already been
                          processed.

      isGapToken   An expected per-message token was
                          not received.

   "true" return value for any of the above methods indicates that the
   token exhibited the specified property. The application must
   determine the appropriate course of action for these supplementary
   values. They are not treated as errors by the GSS-API.

4.13. **Names**

   A name is used to identify a person or entity. GSS-API authenticates
   the relationship between a name and the entity claiming the name.

   Since different authentication mechanisms may employ different
   namespaces for identifying their principals, GSS-API's naming support
   is necessarily complex in multi-mechanism environments (or even in
   some single-mechanism environments where the underlying mechanism
   supports multiple namespaces).

   Two distinct conceptual representations are defined for names:

   1)   A GSS-API form represented by implementations of the GSSName
        interface: A single GSSName object may contain multiple names
        from different namespaces, but all names should refer to the
        same entity. An example of such an internal name would be the
        name returned from a call to the getName method of the
        GSSCredential interface, when applied to a credential containing
        credential elements for multiple authentication mechanisms
        employing different namespaces. This GSSName object will
        contain a distinct name for the entity for each authentication
        mechanism.

   For GSS-API implementations supporting multiple namespaces,
   GSSName implementations must contain sufficient information to
   determine the namespace to which each primitive name belongs.

2)   Mechanism-specific contiguous byte array and string forms:
     Different GSSName initialization methods are provided to handle
     both byte array and string formats and to accommodate various
     calling applications and name types. These formats are capable
     of containing only a single name (from a single namespace).
     Contiguous string names are always accompanied by an object
     identifier specifying the namespace to which the name belongs,
     and their format is dependent on the authentication mechanism
     that employs that name. The string name forms are assumed to be
     printable, and may therefore be used by GSS-API applications for
     communication with their users. The byte array name formats are
     assumed to be in non-printable formats (e.g. the byte array
     returned from the export method of the GSSName interface).

An GSSName object can be converted to a contiguous representation by
using the toString method. This will guarantee that the name will be
converted to a printable format. Different initialization methods in
the GSSName interface are defined allowing support for multiple
syntaxes for each supported namespace, and allowing users the freedom
to choose a preferred name representation. The toString method
should use an implementation-chosen printable syntax for each
supported name-type. To obtain the printable name type,
getStringNameType method can be used.

There is no guarantee that calling the toString method on the GSSName
interface will produce the same string form as the original imported
string name. Furthermore, it is possible that the name was not even
constructed from a string representation. The same applies to name-
space identifiers which may not necessarily survive unchanged after a
journey through the internal name-form. An example of this might be
a mechanism that authenticates X.500 names, but provides an
algorithmic mapping of Internet DNS names into X.500. That
mechanism's implementation of GSSName might, when presented with a
DNS name, generate an internal name that contained both the original
DNS name and the equivalent X.500 name. Alternatively, it might only
store the X.500 name. In the latter case, the toString method of
GSSName would most likely generate a printable X.500 name, rather
than the original DNS name.

The context acceptor can obtain an GSSName object representing the
entity performing the context initiation (through the usage of
getSrcName method). Since this name has been authenticated by a
single mechanism, it contains only a single name (even if the
internal name presented by the context initiator to the GSSContext

object had multiple components). Such names are termed internal
mechanism names, or "MN"s and the names emitted by GSSContext
interface in the getSrcName and getTargName are always of this type.
Since some applications may require MNs without wanting to incur the
overhead of an authentication operation, creation methods are
provided that take not only the name buffer and name type, but also
the mechanism oid for which this name should be created. When
dealing with an existing GSSName object, the canonicalize method may
be invoked to convert a general internal name into an MN.

GSSName objects can be compared using their equal method, which
returns "true" if the two names being compared refer to the same
entity. This is the preferred way to perform name comparisons
instead of using the printable names that a given GSS-API
implementation may support. Since GSS-API assumes that all primitive
names contained within a given internal name refer to the same
entity, equal can return "true" if the two names have at least one
primitive name in common. If the implementation embodies knowledge
of equivalence relationships between names taken from different
namespaces, this knowledge may also allow successful comparisons of
internal names containing no overlapping primitive elements.

When used in large access control lists, the overhead of creating an
GSSName object on each name and invoking the equal method on each
name from the ACL may be prohibitive. As an alternative way of
supporting this case, GSS-API defines a special form of the
contiguous byte array name which may be compared directly (byte by
byte). Contiguous names suitable for comparison are generated by the
export method. Exported names may be re-imported by using the byte
array constructor and specifying the NT_EXPORT_NAME as the name type
object identifier. The resulting GSSName name will also be a MN.
The GSSName interface defines public static Oid objects representing
the standard name types. Structurally, an exported name object
consists of a header containing an OID identifying the mechanism that
authenticated the name, and a trailer containing the name itself,
where the syntax of the trailer is defined by the individual
mechanism specification. Detailed description of the format is
specified in the language-independent GSS-API specification
[GSSAPIv2].

Note that the results obtained by using the equals method will in
general be different from those obtained by invoking canonicalize and
export, and then comparing the byte array output. The first series
of operation determines whether two (unauthenticated) names identify
the same principal; the second whether a particular mechanism would
authenticate them as the same principal. These two operations will
in general give the same results only for MNs.

   It is important to note that the above are guidelines as how GSSName
   implementations should behave, and are not intended to be specific
   requirements of how names objects must be implemented. The mechanism
   designers are free to decide on the details of their implementations
   of the GSSName interface as long as the behavior satisfies the above
   guidelines.

4.14.   **Channel Bindings**

   GSS-API supports the use of user-specified tags to identify a given
   context to the peer application. These tags are intended to be used
   to identify the particular communications channel that carries the
   context. Channel bindings are communicated to the GSS-API using the
   ChannelBinding object. The application may use byte arrays to
   specify the application data to be used in the channel binding as
   well as using instances of the InetAddress. The InetAddress for the
   initiator and/or acceptor can be used within an instance of a
   ChannelBinding. ChannelBinding can be set for the GSSContext object
   using the setChannelBinding method before the first call to init or
   accept has been performed. Unless the setChannelBinding method has
   been used to set the ChannelBinding for an GSSContext object, "null"
   ChannelBinding will be assumed. InetAddress is currently the only
   address type defined within the Java platform and as such, it is the
   only one supported within the ChannelBinding class. Applications
   that use other types of addresses can include them as part of the
   application specific data.

   Conceptually, the GSS-API concatenates the initiator and acceptor
   address information, and the application supplied byte array to form
   an octet string. The mechanism calculates a MIC over this octet
   string and binds the MIC to the context establishment token emitted
   by init method of the GSSContext interface. The same bindings are
   set by the context acceptor for its GSSContext object and during
   processing of the accept method a MIC is calculated in the same way.
   The calculated MIC is compared with that found in the token, and if
   the MICs differ, accept will throw a GSSException with the major
   code set to BAD_BINDINGS, and the context will not be established.
   Some mechanisms may include the actual channel binding data in the
   token (rather than just a MIC); applications should therefore not use
   confidential data as channel-binding components.

   Individual mechanisms may impose additional constraints on addresses
   that may appear in channel bindings. For example, a mechanism may
   verify that the initiator address field of the channel binding
   contains the correct network address of the host system. Portable
   applications should therefore ensure that they either provide correct
   information for the address fields, or omit setting of the addressing

information.

4.15. Stream Objects

The context object provides overloaded methods which use input and output streams as the means to convey authentication and per-message GSS-API tokens. It is important to note that the streams are expected to contain the usual GSS-API tokens which would otherwise be handled through the usage of byte arrays. The tokens are expected to have a definite start and an end. The callers are responsible for ensuring that the supplied streams will not block, or expect to block until a full token is processed by the GSS-API method. Only a single GSS-API token will be processed per invocation of the stream based method.

The usage of streams allows the callers to have control and management of the supplied buffers. Because streams are non-primitive objects, the callers can make the streams as complicated or as simple as desired simply by using the streams defined in the java.io package or creating their own through the use of inheritance. This will allow for the application's greatest flexibility.

4.16. Optional Parameters

Whenever the application wishes to omit an optional parameter the "null" value shall be used. The detailed method descriptions indicate which parameters are optional. Methods overloading has also been used as a technique to indicate default parameters.

5. Introduction to GSS-API Classes and Interfaces

This section presents a brief description of the classes and interfaces that constitute the GSS-API. The implementations of these are obtained from the CLASSPATH defined by the application. If Java GSS becomes part of the standard Java API's then these classes will be available by default on all systems as part of the JRE's system classes.

This section also shows the corresponding RFC 2078 functionality implemented by each of the classes. Detailed description of these classes and their methods is presented in section 6.

5.1. GSSManager class

   This class serves as a factory to instantiate implementations of the
   GSS-API interfaces and also provides methods to make queries about
   underlying security mechanisms. How this class locates and
   communicates with components from the underlying security mechanisms
   is outside the scope of this document, but may be specified in
   whatever SPI the GSSManager implementation follows.

   Applications that desire to provide their own implementation of the
   GSSManager class can simply extend the standard GSSManager class and
   instantiate the new subclass. However, it is recommended that rather
   than replacing the implementation, they enable the existing
   implementation to support their needs by providing the necessary
   components via the SPI that it follows.

   This class contains equivalents of the following RFC 2078 routines:

RFC 2078 Routine	Function	Section
gss_import_name	Create an internal name from the supplied information.	6.1.9-6.1.12
gss_acquire_cred	Acquire credential for use.	6.1.13-6.1.15
gss_import_sec_context	Create a previously exported context.	6.1.18
gss_indicate_mechs	List the mechanisms supported by this GSS-API implementation.	6.1.6
gss_inquire_mechs_for_name	List the mechanisms supporting the specified name type.	6.1.8
gss_inquire_names_for_mech	List the name types supported by the specified mechanism.	6.1.7

5.2. GSSName interface

   GSS-API names are represented in the Java bindings through the

GSSName interface. Different name formats and their definitions are
identified with universal Object Identifiers (oids). The format of
the names can be derived based on the unique oid of each name type.
The following GSS-API routines are provided by the GSSName interface:

RFC 2078 Routine	Function	Section(s)
gss_display_name	Covert internal name representation to text format.	6.2.7
gss_compare_name	Compare two internal names.	6.2.3, 6.2.4
gss_release_name	Release resources associated with the internal name.	N/A
gss_canonicalize_name	Convert an internal name to a mechanism name.	6.1.11,
gss_export_name	Convert a mechanism name to export format.	6.2.6
gss_duplicate_name	Create a copy of the internal name.	N/A

The gss_release_name call is not provided as Java does its own
garbage collection. The gss_duplicate_name call is also redundant;
the GSSName interface has no mutator methods that can change the
state of the object so it is safe for sharing.

5.3. GSSCredential interface

The GSSCredential interface is responsible for the encapsulation of
GSS-API credentials. Credentials identify a single entity and
provide the necessary cryptographic information to enable the
creation of a context on behalf of that entity. A single credential
may contain multiple mechanism specific credentials, each referred to
as a credential element. The GSSCredential interface provides the
functionality of the following GSS-API routines:

RFC 2078 Routine	Function	Section(s)
gss_add_cred	Constructs credentials incrementally.	6.3.12

```
gss_inquire_cred          Obtain information about   6.3.4,6.3.5
                          credential.

gss_inquire_cred_by_mech  Obtain per-mechanism       6.3.5-6.3.10
                          information about
                          a credential.

gss_release_cred          Disposes of credentials       6.3.3
                          after use.
```

5.4. GSSContext interface

This interface encapsulates the functionality of context-level calls
required for security context establishment and management between
peers as well as the per-message services offered to applications. A
context is established between a pair of peers and allows the usage
of security services on a per-message basis on application data. It
is created over a single security mechanism. The GSSContext
interface provides the functionality of the following GSS-API
routines:

```
    RFC 2078 Routine                Function              Section(s)

gss_init_sec_context      Initiate the creation of a      6.4.3,
                          security context with a peer.   6.4.4

gss_accept_sec_context    Accept a security context       6.4.5,
                          initiated by a peer.            6.4.6

gss_delete_sec_context    Destroy a security context.     6.4.8

gss_context_time          Obtain remaining context        6.4.37
                          time.

gss_inquire_context       Obtain context                  6.4.29 to
                          characteristics.                6.3.42

gss_wrap_size_limit       Determine token-size limit      6.4.9
                          for gss_wrap.

gss_export_sec_context    Transfer security context       6.4.18
                          to another process.
```

   gss_get_mic   Calculate a cryptographic   6.4.14,
                          Message Integrity Code (MIC)   6.4.15
                          for a message.

   gss_verify_mic   Verify integrity on a received   6.4.16,
                          message.   6.4.17

   gss_wrap   Attach a MIC to a message and   6.4.10,
                          optionally encrypt the message   6.4.11
                          content.

   gss_unwrap   Obtain a previously wrapped   6.4.12,
                          application message verifying   6.4.13
                          its integrity and optionally
                          decrypting it.

   The functionality offered by the gss_process_context_token routine
   has not been included in the Java bindings specification. The
   corresponding functionality of gss_delete_sec_context has also been
   modified to not return any peer tokens. This has been proposed in
   accordance to the recommendations stated in the RFC 2078 update
   draft. GSSContext does offer the functionality of destroying the
   locally-stored context information.

5.5. MessageProp class

   This helper class is used in the per-message operations on the
   context. An instance of this class is created by the application and
   then passed into the per-message calls. In some cases, the
   application conveys information to the GSS-API implementation through
   this object and in other cases the GSS-API returns information to the
   application by setting it in this object. See the description of the
   per-message operations wrap, unwrap, getMIC, and verifyMIC in the
   GSSContext interfaces for details.

5.6. GSSException class

   Exceptions are used in the Java bindings to signal fatal errors to
   the calling applications. This replaces the major and minor codes
   used in the C-bindings specification as a method of signaling
   failures. The GSSException class handles both minor and major codes,
   as well as their translation into textual representation. All GSS-
   API methods are declared as throwing this exception.

      RFC 2078 Routine   Function   Section

   gss_display_status   Retrieve textual   6.8.5, 6.8.6,
                         representation of error   6.8.8, 6.8.9
                         codes.

5.7. Oid class

   This utility class is used to represent Universal Object Identifiers
   and their associated operations. GSS-API uses object identifiers to
   distinguish between security mechanisms and name types. This class,
   aside from being used whenever an object identifier is needed,
   implements the following GSS-API functionality:

      RFC 2078 Routine   Function   Section

   gss_test_oid_set_member   Determine if the specified oid   6.7.5
                              is part of a set of oids.

5.8. ChannelBinding class

   An instance of this class is used to specify channel binding
   information to the GSSContext object before the start of a security
   context establishment. The application may use a byte array to
   specify application data to be used in the channel binding as well as
   use instances of the InetAddress. InetAddress is currently the only
   address type defined within the Java platform and as such, it is the
   only one supported within the ChannelBinding class. Applications that
   use other types of addresses can include them as part of the
   application data.

6. Detailed GSS-API Class Description

   This section lists a detailed description of all the public methods
   that each of the GSS-API classes and interfaces must provide.

6.1. public class GSSManager

   The GSSManager is a concrete class that serves as a factory for three
   GSS interfaces: GSSName, GSSCredential, and GSSContext. It also

   provides methods for applications to determine what mechanisms are
   available on the system and what nametypes these mechanisms support.

   The GSSManager class may be internally implemented as a modular
   provider based layer that follows some service provider specification
   to lookup underlying mechanism components. It provides the
   application with a method to set a preferred provider for any given
   mechanism.

6.1.1. **Example Code**


```
GSSManager mgr = new GSSManager();

// What mechs are available to us?
Oid[] supportedMechs = mgr.getMechs();

// Set a preference for the provider to be used when support is need
// for the mechanisms "1.2.840.113554.1.2.2" and "1.3.6.1.5.5.2".
// Assume this provider (called "Sun") has already been installed on
// the system.

Oid krb = new Oid("1.2.840.113554.1.2.2");
Oid spnego = new Oid("1.3.6.1.5.5.2");

mgr.setProvider(krb, "Sun");
mgr.setProvider(spnego, "Sun");

// Add support for a new mechanism "1.3.6.1.5.5.1.1" that is
// available from a specific provider being bundled with the
// application.

Oid spkm1 = new Oid("1.3.6.1.5.5.1.1");

Provider prov = (Provider) (new com.foo.security.Provider());

mgr.setProvider(spkm1, prov);

// What name types does this spkm implementation support?
Oid[] nameTypes = mgr.getNamesForMech(spkm1);
```


6.1.2. **Constructors**

```
public GSSManager()
```

   Constructor which instantiates a GSSManager for the application.

6.1.3. **setProvider**

   public void setProvider(Oid mech, String provName)
           throws GSSException, java.security.NoSuchProviderException

   Sets the provider for the GSS-API to use internally when it needs
   components to support the indicated mechanism.

   If the GSSManager implementation does not support an SPI with a
   pluggable provider architecture it should throw a GSSException with
   the status code GSSException.UNAVAILABLE to indicate that the
   operation is unavailable. If the requested provider does not support
   the indicated mech then it should throw a GSSException with the
   status code GSSException.BAD_MECH.

   This call assumes that the provider defined by the string provName is
   already installed on the system either statically or dynamically
   using the Java Cryptography Architecture (JCA) specification. If it
   does not find the provider it should throw a NoSuchProviderException.

   Parameters:

        mech   The mechanism for which the provider is being set

        provName   The name of the provider that should be used whenever
                   support is needed for mech. A value of null will
                   remove any previous preference set by the application.

6.1.4. **setProvider**

   public void setProvider(Oid mech, java.security.Provider p)
           throws GSSException

   Sets the provider for the GSS-API to use internally when it needs
   components to support the indicated mechanism.

   If the GSSManager implementation does not support an SPI with a
   pluggable provider architecture it should throw a GSSException with
   the status code GSSException.UNAVAILABLE to indicate that the
   operation is unavailable. If the requested provider does not support
   the indicated mech then it should throw a GSSException with the
   status code GSSException.BAD_MECH.

   This provider need not be installed globally on the system at all.
   The application instantiates the provider and passes it to the
   instance of GSSManager through this method.

      Parameters:

         mech   The mechanism for which the provider is being set

         p   The provider instance to be used to implement the
                    services for the specified mechanism. A value of null
                    will remove any previous preference set by the
                    application.

6.1.5. getProvider

   public Provider getProvider(Oid mech)

   Returns the preferred Provider instance that the GSS-API would use
   when trying to instantiate elements for the indicated mechanism.

6.1.6. getMechs

   public Oid[] getMechs()

   Returns an array of Oid objects indicating mechanisms available to
   GSS-API callers. A "null" value is returned when no mechanism are
   available (an example of this would be when mechanism are dynamically
   configured, and currently no mechanisms are installed).

6.1.7. getNamesForMech

   public Oid[] getNamesForMech(Oid mech)
                        throws GSSException

   Returns name type Oid's supported by the specified mechanism.

      Parameters:

         mech   The Oid object for the mechanism to query.

6.1.8. **getMechsForName**

    public Oid[] getMechsForName(Oid nameType)

    Returns an array of Oid objects corresponding to the mechanisms that
    support the specific name type. "null" is returned when no
    mechanisms are found to support the specified name type.

    Parameters:

         nameType The Oid object for the name type.

6.1.9. **createName**

    public GSSName createName(String nameStr, Oid nameSpace)
                    throws GSSException

    Factory method to convert a contiguous string name from the specified
    namespace to an GSSName object. In general, the GSSName object
    created will not be an MN; two examples that are exceptions to this
    are when the namespace type parameter indicates NT_EXPORT_NAME or
    when the GSS-API implementation is not multi-mechanism.

    Parameters:

         nameStr   The string representing a printable form of the name
                   to create.

         nameType The Oid specifying the namespace of the printable name
                   supplied. Note that nameType serves to describe and
                   qualify the interpretation of the input nameStr, it
                   does not necessarily imply a type for the output
                   GSSName implementation. "null" value can be used to
                   specify that a mechanism specific default printable
                   syntax should be assumed by each mechanism that
                   examines nameStr.

6.1.10. **createName**

    public GSSName createName(byte name[], Oid nameType)
                    throws GSSException

    Factory method to convert a contiguous byte array containing a name

from the specified namespace to an GSSName object. In general, the
GSSName object created will not be an MN; two examples that are
exceptions to this are when the namespace type parameter indicates
NT_EXPORT_NAME or when the GSS-API implementation is not multi-
mechanism.

Parameters:

    name   The byte array containing the name to create.

    nameType   The Oid specifying the namespace of the name supplied
           in the byte array. Note that nameType serves to
           describe and qualify the interpretation of the input
           name byte array, it does not necessarily imply a type
           for the output GSSName implementation. "null" value
           can be used to specify that a mechanism specific
           default syntax should be assumed by each mechanism
           that examines the byte array.

6.1.11. **createName**

   public GSSName createName(String nameStr, Oid nameType,
                 Oid mechType) throws GSSException

Factory method to convert a contiguous string name from the specified
namespace to an GSSName object that is a mechanism name (MN). In
other words, this method is a utility that does the equivalent of two
steps: the createName described in 6.1.7 and then also the
GSSName.canonicalize() described in 6.2.5.

Parameters:

    nameStr   The string representing a printable form of the name
           to create.

    nameType   The Oid specifying the namespace of the printable name
           supplied. Note that nameType serves to describe and
           qualify the interpretation of the input nameStr, it
           does not necessarily imply a type for the output
           GSSName implementation. "null" value can be used to
           specify that a mechanism specific default printable
           syntax should be assumed when the mechanism examines
           nameStr.

    mechType   Oid specifying the mechanism for which this name
           should be created.

6.1.12. **createName**

   public createName(byte name[], Oid nameType, Oid mechType)
                  throws GSSException

   Factory method to convert a contiguous byte array containing a name
   from the specified namespace to an GSSName object that is an MN. In
   other words, this method is a utility that does the equivalent of two
   steps: the createName described in 6.1.8 and then also the
   GSSName.canonicalize() described in 6.2.5.

   Parameters:

        name   The byte array representing the name to create.

        nameType   The Oid specifying the namespace of the name supplied
                   in the byte array. Note that nameType serves to
                   describe and qualify the interpretation of the input
                   name byte array, it does not necessarily imply a type
                   for the output GSSName implementation. "null" value
                   can be used to specify that a mechanism specific
                   default syntax should be assumed by each mechanism
                   that examines the byte array.

        mechType   Oid specifying the mechanism for which this name
                   should be created.

6.1.13. **createCredential**

   public GSSCredential createCredential (int usage)
                  throws GSSException

   Factory method for acquiring default credentials. This will cause
   the GSS-API to use system specific defaults for the set of
   mechanisms, name, and an INDEFINITE lifetime.

   Parameters:

        usage   The intended usage for this credential object. The
                   value of this parameter must be one of:
                   GSSCredential.ACCEPT_AND_INITIATE,
                   GSSCredential.ACCEPT_ONLY, GSSCredential.INITIATE_ONLY

6.1.14. **createCredential**

   public GSSCredential createCredential (GSSName aName,

```
                int lifetime, Oid mechOid, int usage)
                throws GSSException
```

Factory method for acquiring a single mechanism credential.

Parameters:

    aName   Name of the principal for whom this credential is to
              be acquired. Use "null" to specify the default
              principal.

    lifetime The number of seconds that credentials should remain
              valid. Use GSSCredential.INDEFINITE to request that
              the credentials have the maximum permitted lifetime.

    mechOid   The oid of the desired mechanism. Use "(Oid) null" to
              request the default mechanism(s).

    usage   The intended usage for this credential object. The
              value of this parameter must be one of:
              GSSCredential.ACCEPT_AND_INITIATE,
              GSSCredential.ACCEPT_ONLY, GSSCredential.INITIATE_ONLY

6.1.15. createCredential

```
public GSSCredential createCredential(GSSName aName,
                int lifetime, Oid mechs[], int usage)
                throws GSSException
```

Factory method for acquiring credentials over a set of mechanisms.
Acquires credentials for each of the mechanisms specified in the
array called mechs. To determine the list of mechanisms' for which
the acquisition of credentials succeeded, the caller should use the
GSSCredential.getMechs() method.

Parameters:

    aName   Name of the principal for whom this credential is to
              be acquired. Use "null" to specify the default
              principal.

    lifetime The number of seconds that credentials should remain
              valid. Use GSSCredential.INDEFINITE to request that
              the credentials have the maximum permitted lifetime.

    mechOid   The array of mechanisms over which the credential is
              to be acquired. Use "(Oid[]) null" for requesting a

                 system specific default set of mechanisms.

        usage   The intended usage for this credential object. The
                    value of this parameter must be one of:
                    GSSCredential.ACCEPT_AND_INITIATE,
                    GSSCredential.ACCEPT_ONLY, GSSCredential.INITIATE_ONLY

6.1.16. createContext

```
public GSSContext createContext(GSSName peer, Oid mechOid,
                GSSCredential myCred, int lifetime)
                 throws GSSException
```

Factory method for creating a context on the initiator's side.
Context flags may be modified through the mutator methods prior to
calling GSSContext.initSecContext().

Parameters:

        peer   Name of the target peer.

        mechOid   Oid of the desired mechanism. Use "(Oid) null" to
                    request default mechanism.

        myCred   Credentials of the initiator. Use "null" to act as a
                    default initiator principal.

        lifetime   The request lifetime, in seconds, for the credential.

6.1.17. createContext

```
public GSSContext createContext(GSSCredential myCred)
                 throws GSSException
```

Factory method for creating a context on the acceptor' side. The
context's properties will be determined from the input token supplied
to the accept method.

Parameters:

        myCred   Credentials for the acceptor. Use "null" to act as a
                    default acceptor principal.

6.1.18. **createContext**

```
public GSSContext createContext(byte [] interProcessToken)
                throws GSSException
```

Factory method for creating a previously exported context. The
context properties will be determined from the input token and can't
be modified through the set methods.

Parameters:

   interProcessToken
            The token previously emitted from the export method.

6.2. **public interface GSSName**

This interface encapsulates a single GSS-API principal entity.
Different name formats and their definitions are identified with
universal Object Identifiers (Oids). The format of the names can be
derived based on the unique oid of its namespace type.

6.2.1. **Example Code**

Included below are code examples utilizing the GSSName interface.
The code below creates a GSSName, converts it to a mechanism name
(MN), performs a comparison, obtains a printable representation of
the name, exports it and then re-imports to obtain a new GSSName.

```
GSSManager mgr = new GSSManager();

// create a host based service name
GSSName name = mgr.createName("service@host",
                GSSName.NT_HOSTBASED_SERVICE);

Oid krb5 = new Oid("1.2.840.113554.1.2.2");

GSSName mechName = name.canonicalize(krb5);

// the above two steps are equivalent to the following
GSSName mechName = mgr.createName("service@host",
                GSSName.NT_HOSTBASED_SERVICE, krb5);

// perform name comparison
if (name.equals(mechName))
```

          print("Names are equals.");

    // obtain textual representation of name and its printable
    // name type
    print(mechName.toString() +
                    mechName.getStringNameType().toString());

    // export and re-import the name
    byte [] exportName = mechName.export();

    // create a new name object from the exported buffer
    GSSName newName = mgr.createName(exportName,
                    GSSName.NT_EXPORT_NAME);

6.2.2. Static Constants

    public static final Oid NT_HOSTBASED_SERVICE

    Oid indicating a host-based service name form. It is used to
    represent services associated with host computers. This name form is
    constructed using two elements, "service" and "hostname", as follows:

        service@hostname

    Values for the "service" element are registered with the IANA. It
    represents the following value: { 1(iso), 3(org), 6(dod),
    1(internet), 5(security), 6(nametypes), 2(gss-host-based-services) }

    public static final Oid NT_USER_NAME

    Name type to indicate a named user on a local system. It represents
    the following value: { iso(1) member-body(2) United States(840)
    mit(113554) infosys(1) gssapi(2) generic(1) user_name(1) }

    public static final Oid NT_MACHINE_UID_NAME

    Name type to indicate a numeric user identifier corresponding to a
    user on a local system. (e.g. Uid). It represents the following
    value: { iso(1) member-body(2) United States(840) mit(113554)
    infosys(1) gssapi(2) generic(1) machine_uid_name(2) }

    public static final Oid NT_STRING_UID_NAME

Name type to indicate a string of digits representing the numeric
user identifier of a user on a local system. It represents the
following value: { iso(1) member-body(2) United States(840)
mit(113554) infosys(1) gssapi(2) generic(1) string_uid_name(3) }

public static final Oid NT_ANONYMOUS

Name type for representing an anonymous entity. It represents the
following value: { 1(iso), 3(org), 6(dod), 1(internet), 5(security),
6(nametypes), 3(gss-anonymous-name) }

public static final Oid NT_EXPORT_NAME

Name type used to indicate an exported name produced by the export
method. It represents the following value: { 1(iso), 3(org), 6(dod),
1(internet), 5(security), 6(nametypes), 4(gss-api-exported-name) }

6.2.3. equals

public boolean equals(GSSName another) throws GSSException

Compares two GSSName objects to determine whether they refer to the
same entity. This method may throw a GSSException when the names
cannot be compared. If either of the names represents an anonymous
entity, the method will return "false".

Parameters:

    another   GSSName object to compare with.

6.2.4. equals

public boolean equals(Object another)

A variation of the equals method described in 6.2.3 that is provided
to override the Object.equals() method that the implementing class
will inherit. The behaviour is exactly the same as that in 6.2.3
except that no GSSException is thrown; instead, false will be
returned in the situation where an error occurs.

Parameters:

    another   GSSName object to compare with.

6.2.5. **canonicalize**

   public GSSName canonicalize(Oid mechOid) throws GSSException

   Creates a mechanism name (MN) from an arbitrary internal name. This
   is equivalent to using the factory methods described in 6.1.9 or
   6.1.10 that take the mechanism name as one of their parameters.

   Parameters:

       mechOid   The oid for the mechanism for which the canonical form
                  of the name is requested.

6.2.6. **export**

   public byte[] export() throws GSSException

   Returns a canonical contiguous byte representation of a mechanism
   name (MN), suitable for direct, byte by byte comparison by
   authorization functions. If the name is not an MN, implementations
   may throw a GSSException with the NAME_NOT_MN status code. If an
   implementation chooses not to throw an exception, it should use some
   system specific default mechanism to canonicalize the name and then
   export it. The format of the header of the output buffer is
   specified in RFC 2078.

6.2.7. **toString**

   public String toString()

   Returns a textual representation of the GSSName object. To retrieve
   the printed name format, which determines the syntax of the returned
   string, the getStringNameType method can be used.

6.2.8. **getStringNameType**

   public Oid getStringNameType() throws GSSException

   Returns the oid representing the type of name returned through the
   toString method. Using this oid, the syntax of the printable name
   can be determined.

6.2.9. isAnonymous

   public boolean isAnonymous()

   Tests if this name object represents an anonymous entity. Returns
   "true" if this is an anonymous name.

6.2.10. isMN

   public boolean isMN()

   Tests if this name object contains only one mechanism element and is
   thus a mechanism name as defined by RFC 2078.

6.3. public interface GSSCredential implements Cloneable

   This interface encapsulates the GSS-API credentials for an entity. A
   credential contains all the necessary cryptographic information to
   enable the creation of a context on behalf of the entity that it
   represents. It may contain multiple, distinct, mechanism specific
   credential elements, each containing information for a specific
   security mechanism, but all referring to the same entity.

   A credential may be used to perform context initiation, acceptance,
   or both.

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

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

   Applications will create a credential object passing the desired

   parameters. The application can then use the query methods to obtain
   specific information about the instantiated credential object
   (equivalent to the gss_inquire routines). When the credential is no
   longer needed, the application should call the dispose (equivalent to
   gss_release_cred) method to release any resources held by the
   credential object and to destroy any cryptographically sensitive
   information.

   Classes implementing this interface also implement the Cloneable
   interface. This indicates the the class will support the clone()
   method that will allow the creation of duplicate credentials. This
   is useful when called just before the add() call to retain a copy of
   the original credential.

6.3.1. **Example Code**

   This example code demonstrates the creation of a GSSCredential
   implementation for a specific entity, querying of its fields, and its
   release when it is no longer needed.


```
GSSManager mgr = new GSSManager();

// start by creating a name object for the entity
GSSName name = mgr.createName("userName", GSSName.NT_USER_NAME);

// now acquire credentials for the entity
GSSCredential cred = mgr.createCredential(name,
                GSSCredential.ACCEPT_ONLY);

// display credential information - name, remaining lifetime,
// and the mechanisms it has been acquired over
print(cred.getName().toString());
print(cred.getRemainingLifetime());

Oid [] mechs = cred.getMechs();
if (mechs != null) {
        for (int i = 0; i < mechs.length; i++)
                print(mechs[i].toString());
}

// release system resources held by the credential
cred.dispose();
```

6.3.2. **Static Constants**

   public static final int INITIATE_AND_ACCEPT

   Credential usage flag requesting that it be able to be used for both
   context initiation and acceptance.

   public static final int INITIATE_ONLY

   Credential usage flag requesting that it be able to be used for
   context initiation only.

   public static final int ACCEPT_ONLY

   Credential usage flag requesting that it be able to be used for
   context acceptance only.

   public static final int INDEFINITE

   A lifetime constant representing indefinite credential lifetime.
   This value must be set to the maximum integer value in Java -
   Integer.MAX_VALUE.

6.3.3. **dispose**

   public void dispose() throws GSSException

   Releases any sensitive information that the GSSCredential object may
   be containing. Applications should call this method as soon as the
   credential is no longer needed to minimize the time any sensitive
   information is maintained.

6.3.4. **getName**

   public GSSName getName() throws GSSException

   Retrieves the name of the entity that the credential asserts.

6.3.5. **getName**

   public GSSName getName(Oid mechOID) throws GSSException

    Retrieves a mechanism name of the entity that the credential asserts.
    Equivalent to calling canonicalize() on the name returned by 7.3.3.

    Parameters:

        mechOID   The mechanism for which information should be
                   returned.

6.3.6. getRemainingLifetime

    public int getRemainingLifetime() throws GSSException

    Returns the remaining lifetime in seconds for a credential. The
    remaining lifetime is the minimum lifetime for any of the underlying
    credential mechanisms. A return value of GSSCredential.INDEFINITE
    indicates that the credential does not expire. A return value of 0
    indicates that the credential is already expired.

6.3.7. getRemainingInitLifetime

    public int getRemainingInitLifetime(Oid mech) throws GSSException

    Returns the remaining lifetime is seconds for the credential to
    remain capable of initiating security contexts under the specified
    mechanism. A return value of GSSCredential.INDEFINITE indicates that
    the credential does not expire for context initiation. A return
    value of 0 indicates that the credential is already expired.

    Parameters:

        mechOID   The mechanism for which information should be
                   returned.

6.3.8. getRemainingAcceptLifetime

    public int getRemainingAcceptLifetime(Oid mech) throws GSSException

    Returns the remaining lifetime is seconds for the credential to
    remain capable of accepting security contexts under the specified
    mechanism. A return value of GSSCredential.INDEFINITE indicates that
    the credential does not expire for context acceptance. A return
    value of 0 indicates that the credential is already expired.

    Parameters:

mechOID   The mechanism for which information should be
           returned.

6.3.9. getUsage

   public int getUsage() throws GSSException

   Returns the credential usage flag. The return value will be one of
   GSSCredential.INITIATE_ONLY, GSSCredential.ACCEPT_ONLY, or
   GSSCredential.INITIATE_AND_ACCEPT.

6.3.10. getUsage

   public int getUsage(Oid mechOID) throws GSSException

   Returns the credential usage flag for the specified credential
   mechanism. The return value will be one of
   GSSCredential.INITIATE_ONLY, GSSCredential.ACCEPT_ONLY, or
   GSSCredential.INITIATE_AND_ACCEPT.

   Parameters:

       mechOID   The mechanism for which information should be
                  returned.

6.3.11. getMechs

   public Oid[] getMechs() throws GSSException

   Returns an array of mechanisms supported by this credential.

6.3.12. add

   public void add(GSSName aName, int initLifetime, int acceptLifetime,
                   Oid mech, int usage) throws GSSException

   Adds a mechanism specific credential-element to an existing
   credential. This method allows the construction of credentials one
   mechanism at a time.

   This routine is envisioned to be used mainly by context acceptors
   during the creation of acceptance credentials which are to be used
   with a variety of clients using different security mechanisms.

   This routine adds the new credential element "in-place". To add the
   element in a new credential, first call clone() to obtain a copy of
   this credential, then call its add() method.

   Parameters:

       aName   Name of the principal for whom this credential is to
                  be acquired. Use "null" to specify the default
                  principal.

       initLifetime
                  The number of seconds that credentials should remain
                  valid for initiating of security contexts. Use
                  GSSCredential.INDEFINITE to request that the
                  credentials have the maximum permitted lifetime.

       acceptLifetime
                  The number of seconds that credentials should remain
                  valid for accepting of security contexts. Use
                  GSSCredential.INDEFINITE to request that the
                  credentials have the maximum permitted lifetime.

       mechOid   The mechanisms over which the credential is to be
                  acquired.

       usage   The intended usage for this credential object. The
                  value of this parameter must be one of:
                  GSSCredential.ACCEPT_AND_INITIATE,
                  GSSCredential.ACCEPT_ONLY, GSSCredential.INITIATE_ONLY

6.3.13. **equals**

   public boolean equals(Object another)

   Tests if this GSSCredential refers to the same entity as the supplied
   object. The two credentials must be acquired over the same
   mechanisms and must refer to the same principal. Returns "true" if
   the two GSSCredentials refer to the same entity; "false" otherwise.

   Parameters:

       another   Another GSSCredential object for comparison.

6.4. **public interface GSSContext**

   This interface encapsulates the GSS-API security context and provides
   the security services (wrap, unwrap, getMIC, verifyMIC) that are
   available over the context. Security contexts are established
   between peers using locally acquired credentials. Multiple contexts
   may exist simultaneously between a pair of peers, using the same or
   different set of credentials. GSS-API functions in a manner
   independent of the underlying transport protocol and depends on its
   calling application to transport its tokens between peers.

   Before the context establishment phase is initiated, the context
   initiator may request specific characteristics desired of the
   established context. These can be set using the set methods. After
   the context is established, the caller can check the actual
   characteristic and services offered by the context using the query
   methods.

   The context establishment phase begins with the first call to the
   init method by the context initiator. During this phase the
   initSecContext and acceptSecContext methods will produce GSS-API
   authentication tokens which the calling application needs to send to
   its peer. If an error occurs at any point, an exception will get
   thrown and the code will start executing in a catch block. If not,
   the normal flow of code continues and the application can make a call
   to the isEstablished() method. If this method returns false it
   indicates that a token is needed from its peer in order to continue
   the context establishment phase. A return value of true signals that
   the local end of the context is established. This may still require
   that a token be sent to the peer, if one is produced by GSS-API.
   During the context establishment phase, the isProtReady() method may
   be called to determine if the context can be used for the per-message
   operations. This allows applications to use per-message operations
   on contexts which aren't fully established.

   After the context has been established or the isProtReady() method
   returns "true", the query routines can be invoked to determine the
   actual characteristics and services of the established context. The
   application can also start using the per-message methods of wrap and
   getMIC to obtain cryptographic operations on application supplied
   data.

   When the context is no longer needed, the application should call
   dispose to release any system resources the context may be using.

. **Example Code**

The example code presented below demonstrates the usage of the
GSSContext interface for the initiating peer. Different operations
on the GSSContext object are presented, including: object
instantiation, setting of desired flags, context establishment, query
of actual context flags, per-message operations on application data,
and finally context deletion.

```java
GSSManager mgr = new GSSManager();

// start by creating the name for a service entity
GSSName targetName = mgr.createName("service@host",
                GSSName.NT_HOSTBASED_SERVICE);

// create a context using default credentials for the above entity
// and the implementation specific default mechanism
GSSContext context = mgr.createContext(targetName,
                null,   /* default mechanism */
                null,   /* default credentials */
                GSSContext.INDEFINITE);

// set desired context options - all others are false by default
context.requestConf(true);
context.requestMutualAuth(true);
context.requestReplayDet(true);
context.requestSequenceDet(true);


// establish a context between peers - using byte arrays
byte []inTok = new byte[0];

try {
        do {
                byte[] outTok = context.initSecContext(inTok, 0,
                                                inTok.length);

                // send the token if present
                if (outTok != null)
                        sendToken(outTok);

                // check if we should expect more tokens
                if (context.isEstablished())
                        break;

                // another token expected from peer
                inTok = readToken();
```

```
        } while (true);

} catch (GSSException e) {
        print("GSSAPI error: " + e.getMessage());
}


// display context information
print("Remaining lifetime in seconds = " + context.getLifetime());
print("Context mechanism = " + context.getMech().toString());
print("Initiator = " + context.getSrcName().toString());
print("Acceptor = " + context.getTargName().toString());

if (context.getConfState())
        print("Confidentiality security service available");

if (context.getIntegState())
        print("Integrity security service available");

// perform wrap on an application supplied message, appMsg,
// using QOP = 0, and requesting privacy service
byte [] appMsg ...

MessageProp mProp = new MessageProp(0, true);

byte []tok = context.wrap(appMsg, 0, appMsg.length, mProp);

if (mProp.getPrivacy())
        print("Message protected with privacy.");

sendToken(tok);


// release the local-end of the context
context.dispose();
```

6.4.2. Static Constants

```
public static final int INDEFINITE
```

A lifetime constant representing indefinite context lifetime. This
value must be set to the maximum integer value in Java -
Integer.MAX_VALUE.

6.4.3. initSecContext

```
public byte[] initSecContext(byte inputBuf[], int offset, int len)
              throws GSSException
```

Called by the context initiator to start the context creation process. This is equivalent to the stream based method except that the token buffers are handled as byte arrays instead of using stream objects. This method may return an output token which the application will need to send to the peer for processing by the accept call. "null" return value indicates that no token needs to be sent to the peer. The application can call isEstablished() to determine if the context establishment phase is complete for this peer. A return value of "false" from isEstablished() indicates that more tokens are expected to be supplied to the initSecContext() method. Note that it is possible that the initSecContext() method return a token for the peer, and isEstablished() return "true" also. This indicates that the token needs to be sent to the peer, but the local end of the context is now fully established.

Upon completion of the context establishment, the available context options may be queried through the get methods.

Parameters:

     inputBuf   Token generated by the peer. This parameter is ignored
            on the first call.

     offset   The offset within the inputBuf where the token begins.

     len   The length of the token within the inputBuf (starting
            at the offset).

6.4.3.1. Example Code

```
// Create a new GSSContext implementation object.
// GSSContext wrapper implements interface GSSContext.
GSSContext context = mgr.createContext(...);


byte []inTok = new byte[0];

try {

        do {
                byte[] outTok = context.initSecContext(inTok, 0,
```

```
                                        inTok.length);

                // send the token if present
                if (outTok != null)
                        sendToken(outTok);


                // check if we should expect more tokens
                if (context.isEstablished())
                        break;


                // another token expected from peer
                inTok = readToken();
        } while (true);

} catch (GSSException e) {
        print("GSSAPI error: " + e.getMessage());
}
```


6.4.4. initSecContext

```
public int initSecContext(InputStream inStream,
              OutputStream outStream) throws GSSException
```

Called by the context initiator to start the context creation
process. This is equivalent to the byte array based method. This
method may write an output token to the outStream, which the
application will need to send to the peer for processing by the
accept call. 0 bytes written to the output stream indicate that no
token needs to be sent to the peer. The application can call
isEstablished() to determine if the context establishment phase is
complete for this peer. A return value of "false" from isEstablished
indicates that more tokens are expected to be supplied to the
initSecContext method. Note that it is possible that the
initSecContext() method return a token for the peer, and
isEstablished() return "true" also. This indicates that the token
needs to be sent to the peer, but the local end of the context is now
fully established.

The GSS-API authentication tokens contain a definitive start and end.
This method will attempt to read one of these tokens per invocation,
and may block on the stream if only part of the token is available.

Upon completion of the context establishment, the available context
options may be queried through the get methods.

Parameters:

inStream   Contains the token generated by the peer. This
           parameter is ignored on the first call.

outStream Output stream where the output token will be written.
           During the final stage of context establishment, there
           may be no bytes written.

6.4.4.1. Example Code


```
// Create a new GSSContext implementation object.
// GSSContext wrapper implements interface GSSContext.
GSSContext context = mgr.createContext(...);

// use standard java.io stream objects
ByteArrayOutputStream os = new ByteArrayOutputStream();
ByteArrayInputStream is = null;


try {

        do {
                context.initSecContext(is, os);

                // send token if present
                if (os.size() > 0)
                        sendToken(os);

                // check if we should expect more tokens
                if (context.isEstablished())
                        break;

                // another token expected from peer
                is = recvToken();

        } while (true);

} catch (GSSException e) {
        print("GSSAPI error: " + e.getMessage());
}
```

6.4.5. acceptSecContext

```
public byte[] acceptSecContext(byte inTok[], int offset, int len)
                throws GSSException
```

Called by the context acceptor upon receiving a token from the peer.
This call is equivalent to the stream based method except that the
token buffers are handled as byte arrays instead of using stream
objects.

This method may return an output token which the application will
need to send to the peer for further processing by the init call.
"null" return value indicates that no token needs to be sent to the
peer. The application can call isEstablished() to determine if the
context establishment phase is complete for this peer. A return
value of "false" from isEstablished() indicates that more tokens are
expected to be supplied to this method.

Note that it is possible that acceptSecContext() return a token for
the peer, and isEstablished() return "true" also. This indicates
that the token needs to be sent to the peer, but the local end of the
context is now fully established.

Upon completion of the context establishment, the available context
options may be queried through the get methods.

Parameters:

    inTok   Token generated by the peer.

    offset   The offset within the inTok where the token begins.

    len   The length of the token within the inTok (starting at
              the offset).

6.4.5.1. Example Code

```
// acquire server credentials
GSSCredential server = mgr.createCredential(...);

// create acceptor GSS-API context fromthe default provider
GSSContext context = mgr.createContext(server, null);

try {
        do {
                byte [] inTok = readToken();
```

```
                byte []outTok = context.acceptSecContext(inTok, 0,
                                        inTok.length);

                // possibly send token to peer
                if (outTok != null)
                        sendToken(outTok);

                // check if local context establishment is complete
                if (context.isEstablished())
                        break;
        } while (true);

} catch (GSSException e) {
        print("GSS-API error: " + e.getMessage());
}
```

. **acceptSecContext**

```
public void acceptSecContext(InputStream inStream,
                OutputStream outStream) throws GSSException
```

Called by the context acceptor upon receiving a token from the peer.
This call is equivalent to the byte array method. It may write an
output token to the outStream, which the application will need to
send to the peer for processing by its initSecContext method. 0 bytes
written to the output stream indicate that no token needs to be sent
to the peer. The application can call isEstablished() to determine
if the context establishment phase is complete for this peer. A
return value of "false" from isEstablished() indicates that more
tokens are expected to be supplied to this method.

Note that it is possible that acceptSecContext() return a token for
the peer, and isEstablished() return "true" also. This indicates
that the token needs to be sent to the peer, but the local end of the
context is now fully established.

The GSS-API authentication tokens contain a definitive start and end.
This method will attempt to read one of these tokens per invocation,
and may block on the stream if only part of the token is available.

Upon completion of the context establishment, the available context
options may be queried through the get methods.

Parameters:

        inStream Contains the token generated by the peer.

outStream Output stream where the output token will be written.
          During the final stage of context establishment, there
          may be no bytes written.

6.4.6.1. **Example Code**


```
// acquire server credentials
GSSCredential server = mgr.createCredential(...);

// create acceptor GSS-API context fromthe default provider
GSSContext context = mgr.createContext(server, null);

// use standard java.io stream objects
ByteArrayOutputStream os = new ByteArrayOutputStream();
ByteArrayInputStream is = null;


try {
      do {

            is = recvToken();

            context.acceptSecContext(is, os);

            // possibly send token to peer
            if (os.size() > 0)
                  sendToken(os);

            // check if local context establishment is complete
            if (context.isEstablished())
                  break;
      } while (true);

} catch (GSSException e) {
      print("GSS-API error: " + e.getMessage());
}
```


6.4.7. **isEstablished**

```
public boolean isEstablished()
```

Used during context establishment to determine the state of the
context. Returns "true" if this is a fully established context on
the caller's side and no more tokens are needed from the peer.

   Should be called after a call to initSecContext() or
   acceptSecContext() when no GSSException is thrown.

6.4.8. dispose

   public void dispose() throws GSSException

   Releases any system resources and cryptographic information stored in
   the context object. This will invalidate the context.

6.4.9. getWrapSizeLimit

   public int getWrapSizeLimit(int qop, boolean confReq,
                   int maxTokenSize) throws GSSException

   Returns the maximum message size that, if presented to the wrap
   method with the same confReq and qop parameters, will result in an
   output token containing no more than the maxTokenSize bytes.

   This call is intended for use by applications that communicate over
   protocols that impose a maximum message size. It enables the
   application to fragment messages prior to applying protection.

   GSS-API implementations are recommended but not required to detect
   invalid QOP values when getWrapSizeLimit is called. This routine
   guarantees only a maximum message size, not the availability of
   specific QOP values for message protection.

   Successful completion of this call does not guarantee that wrap will
   be able to protect a message of the computed length, since this
   ability may depend on the availability of system resources at the
   time that wrap is called. However, if the implementation itself
   imposes an upper limit on the length of messages that may be
   processed by wrap, the implementation should not return a value that
   is greater than this length.

   Parameters:

        qop   Indicates the level of protection wrap will be asked
                   to provide.

        confReq   Indicates if wrap will be asked to provide privacy
                   service.

        maxTokenSize
                   The desired maximum size of the token emitted by wrap.

6.4.10. **wrap**

     public byte[] wrap(byte inBuf[], int offset, int len,
                    MessageProp msgProp) throws GSSException

     Applies per-message security services over the established security
     context. The method will return a token with a cryptographic MIC and
     may optionally encrypt the specified inBuf. This method is
     equivalent in functionality to its stream counterpart. The returned
     byte array will contain both the MIC and the message.

     The MessageProp object is instantiated by the application and used to
     specify a QOP value which selects cryptographic algorithms, and a
     privacy service to optionally encrypt the message. The underlying
     mechanism that is used in the call may not be able to provide the
     privacy service. It sets the actual privacy service that it does
     provide in this MessageProp object which the caller should then query
     upon return. If the mechanism is not able to provide the requested
     QOP, it throws a GSSException with the BAD_QOP code.

     Since some application-level protocols may wish to use tokens emitted
     by wrap to provide "secure framing", implementations should support
     the wrapping of zero-length messages.

     The application will be responsible for sending the token to the
     peer.

     Parameters:

          inBuf   Application data to be protected.

          offset   The offset within the inBuf where the data begins.

          len   The length of the data within the inBuf (starting at
                     the offset).

          msgProp   Instance of MessageProp that is used by the
                     application to set the desired QOP and privacy state.
                     Set the desired QOP to 0 to request the default QOP.
                     Upon return from this method, this object will contain
                     the the actual privacy state that was applied to the
                     message by the underlying mechanism.

6.4.11. **wrap**

     public void wrap(InputStream inStream, OutputStream outStream,
                    MessageProp msgProp) throws GSSException

Allows to apply per-message security services over the established
security context. The method will produce a token with a
cryptographic MIC and may optionally encrypt the message in inStream.
The outStream will contain both the MIC and the message.

The MessageProp object is instantiated by the application and used to
specify a QOP value which selects cryptographic algorithms, and a
privacy service to optionally encrypt the message. The underlying
mechanism that is used in the call may not be able to provide the
privacy service. It sets the actual privacy service that it does
provide in this MessageProp object which the caller should then query
upon return. If the mechanism is not able to provide the requested
QOP, it throws a GSSException with the BAD_QOP code.

Since some application-level protocols may wish to use tokens emitted
by wrap to provide "secure framing", implementations should support
the wrapping of zero-length messages.

The application will be responsible for sending the token to the
peer.

Parameters:

        inStream   Input stream containing the application data to be
                   protected.

        outStream The output stream to write the protected message to.
                   The application is responsible for sending this to the
                   other peer for processing in its unwrap method.

        msgProp   Instance of MessageProp that is used by the
                   application to set the desired QOP and privacy state.
                   Set the desired QOP to 0 to request the default QOP.
                   Upon return from this method, this object will contain
                   the the actual privacy state that was applied to the
                   message by the underlying mechanism.

6.4.12. unwrap

   public byte [] unwrap(byte[] inBuf, int offset, int len,
                    MessageProp msgProp) throws GSSException

   Used by the peer application to process tokens generated with the
   wrap call. This call is equal in functionality to its stream
   counterpart. The method will return the message supplied in the peer
   application to the wrap call, verifying the embedded MIC.

The MessageProp object is instantiated by the application and is used
by the underlying mechanism to return information to the caller such
as the QOP, whether confidentiality was applied to the message, and
other supplementary message state information.

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

   Parameters:

        inBuf   GSS-API wrap token received from peer.

        offset   The offset within the inBuf where the token begins.

        len   The length of the token within the inBuf (starting at
                   the offset).

        msgProp   Upon return from the method, this object will contain
                   the applied QOP, the privacy state of the message, and
                   supplementary information described in 4.12.3 stating
                   whether the token was a duplicate, old, out of
                   sequence or arriving after a gap.

6.4.13. unwrap

   public void unwrap(InputStream inStream, OutputStream outStream,
                   MessageProp msgProp) throws GSSException

   Used by the peer application to process tokens generated with the
   wrap call. This call is equal in functionality to its byte array
   counterpart. It will produce the message supplied in the peer
   application to the wrap call, verifying the embedded MIC.

   The MessageProp object is instantiated by the application and is used
   by the underlying mechanism to return information to the caller such
   as the QOP, whether confidentiality was applied to the message, and
   other supplementary message state information.

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

   Parameters:

        inStream   Input stream containing the GSS-API wrap token
                   received from the peer.

       outStream The output stream to write the application message to.

       msgProp   Upon return from the method, this object will contain
                 the applied QOP, the privacy state of the message, and
                 supplementary information described in 4.12.3 stating
                 whether the token was a duplicate, old, out of
                 sequence or arriving after a gap.

   6.4.14. **getMIC**

    public byte[] getMIC(byte []inMsg, int offset, int len,
                     MessageProp msgProp) throws GSSException

    Returns a token containing a cryptographic MIC for the supplied
    message, for transfer to the peer application. Unlike wrap, which
    encapsulates the user message in the returned token, only the message
    MIC is returned in the output token. This method is identical in
    functionality to its stream counterpart.

    Note that privacy can only be applied through the wrap call.

    Since some application-level protocols may wish to use tokens emitted
    by getMIC to provide "secure framing", implementations should support
    derivation of MICs from zero-length messages.

    Parameters:

       inMsg   Message to generate MIC over.

       offset   The offset within the inMsg where the token begins.

       len   The length of the token within the inMsg (starting at
                 the offset).

       msgProp   Instance of MessageProp that is used by the
                 application to set the desired QOP. Set the desired
                 QOP to 0 in msgProp to request the default QOP.
                 Alternatively pass in "null" for msgProp to request
                 default QOP.

   6.4.15. **getMIC**

    public void getMIC(InputStream inStream, OutputStream outStream,
                     MessageProp msgProp) throws GSSException

    Produces a token containing a cryptographic MIC for the supplied

   message, for transfer to the peer application. Unlike wrap, which
   encapsulates the user message in the returned token, only the message
   MIC is produced in the output token. This method is identical in
   functionality to its byte array counterpart.

   Note that privacy can only be applied through the wrap call.

   Since some application-level protocols may wish to use tokens emitted
   by getMIC to provide "secure framing", implementations should support
   derivation of MICs from zero-length messages.

   Parameters:

        inStream inStream Input stream containing the message to
                  generate MIC over.

        outStream outStream Output stream to write the GSS-API output
                  token to.

        msgProp   Instance of MessageProp that is used by the
                  application to set the desired QOP. Set the desired
                  QOP to 0 in msgProp to request the default QOP.
                  Alternatively pass in "null" for msgProp to request
                  default QOP.

[6.4.16]. **verifyMIC**

   public void verifyMIC(byte []inTok, int tokOffset, int tokLen,
                    byte[] inMsg, int msgOffset, int msgLen,
                    MessageProp msgProp) throws GSSException

   Verifies the cryptographic MIC, contained in the token parameter,
   over the supplied message. This method is equivalent in
   functionality to its stream counterpart.

   The MessageProp object is instantiated by the application and is used
   by the underlying mechanism to return information to the caller such
   as the QOP indicating the strength of protection that was applied to
   the message and other supplementary message state information.

   Since some application-level protocols may wish to use tokens emitted
   by getMIC to provide "secure framing", implementations should support
   the calculation and verification of MICs over zero-length messages.

   Parameters:

        inTok   Token generated by peer's getMIC method.

tokOffset The offset within the inTok where the token begins.

tokLen   The length of the token within the inTok (starting at
          the offset).

inMsg   Application message to verify the cryptographic MIC
          over.

msgOffset The offset within the inMsg where the message begins.

msgLen   The length of the message within the inMsg (starting
          at the offset).

msgProp   Upon return from the method, this object will contain
          the applied QOP and supplementary information
          described in 4.12.3 stating whether the token was a
          duplicate, old, out of sequence or arriving after a
          gap. The confidentiality state will be set to
          "false".

6.4.17. verifyMIC

   public void verifyMIC(InputStream tokStream, InputStream msgStream,
                  MessageProp msgProp) throws GSSException

   Verifies the cryptographic MIC, contained in the token parameter,
   over the supplied message. This method is equivalent in
   functionality to its byte array counterpart.

   The MessageProp object is instantiated by the application and is used
   by the underlying mechanism to return information to the caller such
   as the QOP indicating the strength of protection that was applied to
   the message and other supplementary message state information.

   Since some application-level protocols may wish to use tokens emitted
   by getMIC to provide "secure framing", implementations should support
   the calculation and verification of MICs over zero-length messages.

   Parameters:

        tokStream Input stream containing the token generated by peer's
                  getMIC method.

        msgStream Input stream containing the application message to
                  verify the cryptographic MIC over.

        msgProp   Upon return from the method, this object will contain

                    the applied QOP and supplementary information
                    described in 4.12.3 stating whether the token was a
                    duplicate, old, out of sequence or arriving after a
                    gap. The confidentiality state will be set to
                    "false".

6.4.18. export

   public byte [] export() throws GSSException

   Provided to support the sharing of work between multiple processes.
   This routine will typically be used by the context-acceptor, in an
   application where a single process receives incoming connection
   requests and accepts security contexts over them, then passes the
   established context to one or more other processes for message
   exchange.

   This method deactivates the security context and creates an
   interprocess token which, when passed to the byte array constructor
   of the GSSContext interface in another process, will re-activate the
   context in the second process. Only a single instantiation of a
   given context may be active at any one time; a subsequent attempt by
   a context exporter to access the exported security context will fail.

   The implementation may constrain the set of processes by which the
   interprocess token may be imported, either as a function of local
   security policy, or as a result of implementation decisions. For
   example, some implementations may constrain contexts to be passed
   only between processes that run under the same account, or which are
   part of the same process group.

   The interprocess token may contain security-sensitive information
   (for example cryptographic keys). While mechanisms are encouraged to
   either avoid placing such sensitive information within interprocess
   tokens, or to encrypt the token before returning it to the
   application, in a typical GSS-API implementation this may not be
   possible. Thus the application must take care to protect the
   interprocess token, and ensure that any process to which the token is
   transferred is trustworthy.

6.4.19. requestMutualAuth

   public void requestMutualAuth(boolean state) throws GSSException

   Sets the request state of the mutual authentication flag for the
   context. This method is only valid before the context creation

   process begins and only for the initiator.

   Parameters:

      state   Boolean representing if mutual authentication should
                 be requested during context establishment.

. **requestReplayDet**

   public void requestReplayDet(boolean state) throws GSSException

   Sets the request state of the replay detection service for the
   context. This method is only valid before the context creation
   process begins and only for the initiator.

   Parameters:

      state   Boolean representing if replay detection is desired
                 over the established context.

. **requestSequenceDet**

   public void requestSequenceDet(boolean state) throws GSSException

   Sets the request state for the sequence checking service of the
   context. This method is only valid before the context creation
   process begins and only for the initiator.

   Parameters:

      state   Boolean representing if sequence detection is desired
                 over the established context.

. **requestCredDeleg**

   public void requestCredDeleg(boolean state) throws GSSException

   Sets the request state for the credential delegation flag for the
   context. This method is only valid before the context creation
   process begins and only for the initiator.

   Parameters:

      state   Boolean representing if credential delegation is
                 desired.

6.4.23. **requestAnonymity**

   public void requestAnonymity(boolean state) throws GSSException

   Requests anonymous support over the context. This method is only
   valid before the context creation process begins and only for the
   initiator.

   Parameters:

      state   Boolean representing if anonymity support is
                requested.

6.4.24. **requestConf**

   public void requestConf(boolean state) throws GSSException

   Requests that confidentiality service be available over the context.
   This method is only valid before the context creation process begins
   and only for the initiator.

   Parameters:

      state   Boolean indicating if confidentiality services are to
                be requested for the context.

6.4.25. **requestInteg**

   public void requestInteg(boolean state) throws GSSException

   Requests that integrity services be available over the context. This
   method is only valid before the context creation process begins and
   only for the initiator.

   Parameters:

      state   Boolean indicating if integrity services are to be
                requested for the context.

6.4.26. **requestLifetime**

   public void requestLifetime(int lifetime) throws GSSException

   Sets the desired lifetime for the context in seconds. This method is
   only valid before the context creation process begins and only for

the initiator.

Parameters:

lifetime   The desired context lifetime in seconds.

6.4.27. setChannelBinding

public void setChannelBinding(ChannelBinding cb) throws GSSException

Sets the channel bindings to be used during context establishment.
This method is only valid before the context creation process begins.

Parameters:

cb   Channel bindings to be used.

6.4.28. getCredDelegState

public boolean getCredDelegState()

Returns the state of the delegated credentials for the context. When
issued before context establishment is completed or when the
isProtReady method returns "false", it returns the desired state,
otherwise it will indicate the actual state over the established
context.

6.4.29. getMutualAuthState

public boolean getMutualAuthState()

Returns the state of the mutual authentication option for the
context. When issued before context establishment completes or when
the isProtReady method returns "false", it returns the desired state,
otherwise it will indicate the actual state over the established
context.

6.4.30. getReplayDetState

public boolean getReplayDetState()

Returns the state of the replay detection option for the context.
When issued before context establishment completes or when the
isProtReady method returns "false", it returns the desired state,

otherwise it will indicate the actual state over the established
context.

6.4.31. getSequenceDetState

public boolean getSequenceDetState()

Returns the state of the sequence detection option for the context.
When issued before context establishment completes or when the
isProtReady method returns "false", it returns the desired state,
otherwise it will indicate the actual state over the established
context.

6.4.32. getAnonymityState

public boolean getAnonymityState()

Returns "true" if this is an anonymous context. When issued before
context establishment completes or when the isProtReady method
returns "false", it returns the desired state, otherwise it will
indicate the actual state over the established context.

6.4.33. isTransferable

public boolean isTransferable() throws GSSException

Returns "true" if the context is transferable to other processes
through the use of the export method. This call is only valid on
fully established contexts.

6.4.34. isProtReady

public boolean isProtReady()

Returns "true" if the per message operations can be applied over the
context. Some mechanisms may allow the usage of per-message
operations before the context is fully established. This will also
indicate that the get methods will return actual context state
characteristics instead of the desired ones.

6.4.35. getConfState

public boolean getConfState()

Returns the confidentiality service state over the context. When
issued before context establishment completes or when the isProtReady
method returns "false", it returns the desired state, otherwise it
will indicate the actual state over the established context.

6.4.36. getIntegState

    public boolean getIntegState()

Returns the integrity service state over the context. When issued
before context establishment completes or when the isProtReady method
returns "false", it returns the desired state, otherwise it will
indicate the actual state over the established context.

6.4.37. getLifetime

    public int getLifetime()

Returns the context lifetime in seconds. When issued before context
establishment completes or when the isProtReady method returns
"false", it returns the desired lifetime, otherwise it will indicate
the remaining lifetime for the context.

6.4.38. getSrcName

    public GSSName getSrcName() throws GSSException

Returns the name of the context initiator. This call is valid only
after the context is fully established or the isProtReady method
returns "true". It is guaranteed to return an MN.

6.4.39. getTargName

    public GSSName getTargName() throws GSSException

Returns the name of the context target (acceptor). This call is
valid only after the context is fully established or the isProtReady
method returns "true". It is guaranteed to return an MN.

6.4.40. getMech

    public Oid getMech() throws GSSException

   Returns the mechanism oid for this context.

6.4.41. getDelegCred

   public GSSCredential getDelegCred() throws GSSException

   Returns the delegated credential object on the acceptor's side. To
   check for availability of delegated credentials call
   getDelegCredState. This call is only valid on fully established
   contexts.

6.4.42. isInitiator

   public boolean isInitiator() throws GSSException

   Returns "true" if this is the initiator of the context. This call is
   only valid after the context creation process has started.

6.5. public class MessageProp

   This is a utility class used within the per-message GSSContext
   methods to convey per-message properties.

   When used with the GSSContext interface's wrap and getMIC methods, an
   instance of this class is used to indicate the desired QOP and to
   request if confidentiality services are to be applied to caller
   supplied data (wrap only). To request default QOP, the value of 0
   should be used for QOP.

   When used with the unwrap and verifyMIC methods of the GSSContext
   interface, an instance of this class will be used to indicate the
   applied QOP and confidentiality services over the supplied message.
   In the case of verifyMIC, the confidentiality state will always be
   "false". Upon return from these methods, this object will also
   contain any supplementary status values applicable to the processed
   token. The supplementary status values can indicate old tokens, out
   of sequence tokens, gap tokens or duplicate tokens.

6.5.1. Constructors

   public MessageProp(boolean privState)

   Constructor which sets QOP to 0 indicating that the default QOP is

requested.

Parameters:

   privState The desired privacy state. "true" for privacy and
            "false" for integrity only.

public MessageProp(int qop, boolean privState)

Constructor which sets the values for the qop and privacy state.

Parameters:

   qop   The desired QOP. Use 0 to request a default QOP.

   privState The desired privacy state. "true" for privacy and
            "false" for integrity only.

6.5.2. getQOP

public int getQOP()

Retrieves the QOP value.

6.5.3. getPrivacy

public boolean getPrivacy()

Retrieves the privacy state.

6.5.4. getMinorStatus

public int getMinorStatus()

Retrieves the minor status that the underlying mechanism might have
set.

6.5.5. getMinorString

public String getMinorString()

Returns a string explaining the mechanism specific error code. null
will be returned when no mechanism error code has been set.

[6.5.6](). **setQOP**

   public void setQOP(int qopVal)

   Sets the QOP value.

   Parameters:

       qopVal   The QOP value to be set. Use 0 to request a default
                 QOP value.

[6.5.7](). **setPrivacy**

   public void setPrivacy(boolean privState)

   Sets the privacy state.

   Parameters:

       privState The privacy state to set.

[6.5.8](). **isDuplicateToken**

   public boolean isDuplicateToken()

   Returns "true" if this is a duplicate of an earlier token.

[6.5.9](). **isOldToken**

   public boolean isOldToken()

   Returns "true" if the token's validity period has expired.

[6.5.10](). **isUnseqToken**

   public boolean isUnseqToken()

   Returns "true" if a later token has already been processed.

[6.5.11](). **isGapToken**

   public boolean isGapToken()

   Returns "true" if an expected per-message token was not received.

6.5.12. **setSupplementaryStates**

   public void setSupplementaryStates(boolean duplicate,
                    boolean old, boolean unseq, boolean gap,
                    int minorStatus, String minorString)

   This method sets the state for the supplementary information flags
   and the minor status in MessageProp. It is not used by the
   application but by the GSS implementation to return this information
   to the caller of a per-message context method.

   Parameters:

        duplicate true if the token was a duplicate of an earlier token,
                  false otherwise

        old   true if the token's validity period has expired, false
                  otherwise

        unseq   true if a later token has already been processed,
                  false otherwise

        gap   true if one or more predecessor tokens have not yet
                  been successfully processed, false otherwise

        minorStatus   the integer minor status code that the underlying
                      mechanism wants to set

        minorString   the textual representation of the minorStatus
                      value

6.6. **public class ChannelBinding**

   The GSS-API accommodates the concept of caller-provided channel
   binding information. Channel bindings are used to strengthen the
   quality with which peer entity authentication is provided during
   context establishment. They enable the GSS-API callers to bind the
   establishment of the security context to relevant characteristics
   like addresses or to application specific data.

   The caller initiating the security context must determine the
   appropriate channel binding values to set in the GSSContext object.

The acceptor must provide an identical binding in order to validate
that received tokens possess correct channel-related characteristics.

Use of channel bindings is optional in GSS-API. Since channel-
binding information may be transmitted in context establishment
tokens, applications should therefore not use confidential data as
channel-binding components.

6.6.1. Constructors

```
public ChannelBinding(InetAddress initAddr, InetAddress acceptAddr,
                byte[] appData)
```

Create a ChannelBinding object with user supplied address information
and data. "null" values can be used for any fields which the
application does not want to specify.

Parameters:

    initAddr   The address of the context initiator. "null" value
            can be supplied to indicate that the application does
            not want to set this value.

    acceptAddrThe address of the context acceptor. "null" value can
            be supplied to indicate that the application does not
            want to set this value.

    appData   Application supplied data to be used as part of the
            channel bindings. "null" value can be supplied to
            indicate that the application does not want to set
            this value.


```
public ChannelBinding(byte[] appData)
```

Creates a ChannelBinding object without any addressing information.

Parameters:

    appData   Application supplied data to be used as part of the
            channel bindings.

6.6.2. getInitiatorAddress


```
public InetAddress getInitiatorAddress()
```

Returns the initiator's address for this channel binding. "null" is
returned if the address has not been set.

6.6.3. getAcceptorAddress

public InetAddress getAcceptorAddress()

Returns the acceptor's address for this channel binding. "null" is
returned if the address has not been set.

6.6.4. getApplicationData

public byte[] getApplicationData()

Returns application data being used as part of the ChannelBinding.
"null" is returned if no application data has been specified for the
channel binding.

6.6.5. equals

public boolean equals(Object obj)

Returns "true" if two channel bindings match.

Parameters:

   obj   Another channel binding to compare with.

6.7. public class Oid

This class represents Universal Object Identifiers (Oids) and their
associated operations.

Oids are hierarchically globally-interpretable identifiers used
within the GSS-API framework to identify mechanisms and name formats.

The structure and encoding of Oids is defined in ISOIEC-8824 and
ISOIEC-8825. For example the Oid representation of Kerberos V5
mechanism is "1.2.840.113554.1.2.2"

The GSSName name class contains public static Oid objects

representing the standard name types defined in GSS-API.

6.7.1. **Constructors**

public Oid(String strOid) throws GSSException

Creates an Oid object from a string representation of its integer
components (e.g. "1.2.840.113554.1.2.2").

Parameters:

strOid   The string representation for the oid.

public Oid(InputStream derOid) throws GSSException

Creates an Oid object from its DER encoding. This refers to the full
encoding including tag and length. The structure and encoding of
Oids is defined in ISOIEC-8824 and ISOIEC-8825. This method is
identical in functionality to its byte array counterpart.

Parameters:

derOid   Stream containing the DER encoded oid.

public Oid(byte[] DEROid) throws GSSException

Creates an Oid object from its DER encoding. This refers to the full
encoding including tag and length. The structure and encoding of
Oids is defined in ISOIEC-8824 and ISOIEC-8825. This method is
identical in functionality to its byte array counterpart.

Parameters:

derOid   Byte array storing a DER encoded oid.

6.7.2. **toString**

public String toString()

Returns a string representation of the oid's integer components in
dot separated notation (e.g. "1.2.840.113554.1.2.2").

[6.7.3](). **equals**

   public boolean equals(Object Obj)

   Returns "true" if the two Oid objects represent the same oid value.

   Parameters:

        obj   Another Oid object to compare with.

[6.7.4](). **getDER**

   public byte[] getDER()

   Returns the full ASN.1 DER encoding for this oid object, which
   includes the tag and length.

[6.7.5](). **containedIn**

   public boolean containedIn(Oid[] oids)

   A utility method to test if an Oid object is contained within the
   supplied Oid object array.

   Parameters:

        oids   An array of oids to search.

[6.8](). **public class GSSException extends Exception**

   This exception is thrown whenever a fatal GSS-API error occurs
   including mechanism specific errors. It may contain both, the major
   and minor, GSS-API status codes. The mechanism implementers are
   responsible for setting appropriate minor status codes when throwing
   this exception. Aside from delivering the numeric error code(s) to
   the caller, this class performs the mapping from their numeric values
   to textual representations. All Java GSS-API methods are declared
   throwing this exception.

   All implementations are encouraged to use the Java
   internationalization techniques to provide local translations of the
   message strings.

[6.8.1](#). **Static Constants**

All valid major GSS-API error code values are declared as constants
in this class.

public static final int BAD_BINDINGS

Channel bindings mismatch error.

public static final int BAD_MECH

Unsupported mechanism requested error.

public static final int BAD_NAME

Invalid name provided error.

public static final int BAD_NAMETYPE

Name of unsupported type provided error.

public static final int BAD_STATUS

Invalid status code error - this is the default status value.

public static final int BAD_MIC

Token had invalid integrity check error.

public static final int CONTEXT_EXPIRED

Specified security context expired error.

public static final int CREDENTIALS_EXPIRED

Expired credentials detected error.

public static final int DEFECTIVE_CREDENTIAL

Defective credential error.

public static final int DEFECTIVE_TOKEN

Defective token error.

public static final int FAILURE

General failure, unspecified at GSS-API level.

public static final int NO_CONTEXT

Invalid security context error.

public static final int NO_CRED

Invalid credentials error.

public static final int BAD_QOP

Unsupported QOP value error.

public static final int UNAUTHORIZED

Operation unauthorized error.

public static final int UNAVAILABLE

Operation unavailable error.

public static final int DUPLICATE_ELEMENT

Duplicate credential element requested error.

public static final int NAME_NOT_MN

Name contains multi-mechanism elements error.

   public static final int DUPLICATE_TOKEN

   The token was a duplicate of an earlier token. This is a fatal error
   code that may occur during context establishment. It is not used to
   indicate supplementary status values. The MessageProp object is used
   for that purpose.

   public static final int OLD_TOKEN

   The token's validity period has expired. This is a fatal error code
   that may occur during context establishment. It is not used to
   indicate supplementary status values. The MessageProp object is used
   for that purpose.

   public static final int UNSEQ_TOKEN

   A later token has already been processed. This is a fatal error code
   that may occur during context establishment. It is not used to
   indicate supplementary status values. The MessageProp object is used
   for that purpose.

   public static final int GAP_TOKEN

   An expected per-message token was not received. This is a fatal
   error code that may occur during context establishment. It is not
   used to indicate supplementary status values. The MessageProp object
   is used for that purpose.

6.8.2. Constructors

   public GSSException(int majorCode)

   Creates a GSSException object with a specified major code.

   Parameters:

      majorCode The GSS error code causing this exception to be
                thrown.

   public GSSException(int majorCode, int minorCode, String minorString)

   Creates a GSSException object with the specified major code, minor
   code, and minor code textual explanation. This constructor is to be

used when the exception is originating from the security mechanism.
It allows to specify the GSS code and the mechanism code.

Parameters:

majorCode   The GSS error code causing this exception to be
                thrown.

minorCode   The mechanism error code causing this exception
                to be thrown.

minorString   The textual explanation of the mechanism error
                code.

6.8.3. getMajor

public int getMajor()

Returns the major code representing the GSS error code that caused
this exception to be thrown.

6.8.4. getMinor

public int getMinor()

Returns the mechanism error code that caused this exception. The
minor code is set by the underlying mechanism. Value of 0 indicates
that mechanism error code is not set.

6.8.5. getMajorString

public String getMajorString()

Returns a string explaining the GSS major error code causing this
exception to be thrown.

6.8.6. getMinorString

public String getMinorString()

Returns a string explaining the mechanism specific error code. null
will be returned when no mechanism error code has been set.

6.8.7. **setMinor**

   public void setMinor(int minorCode, String message)

   Used internally by the GSS-API implementation and the underlying
   mechanisms to set the minor code and its textual representation.

   Parameters:

      minorCode The mechanism specific error code.

      message   A textual explanation of the mechanism error code.

6.8.8. **toString**

   public String toString()

   Returns a textual representation of both the major and minor status
   codes.

6.8.9. **getMessage**

   public String getMessage()

   Returns a detailed message of this exception. Overrides
   Throwable.getMessage. It is customary in Java to use this method to
   obtain exception information.

7. **Sample Applications**

   Full Copyright Statement

      Copyright (C) The Internet Society (1999). All Rights Reserved.

      This document and translations of it may be copied and furnished
   to others, and derivative works that comment on or otherwise explain
   it or assist in its implementation may be prepared, copied, published
   and distributed, in whole or in part, without restriction of any
   kind, provided that the above copyright notice and this paragraph are
   included on all such copies and derivative works. However, this
   document itself may not be modified in any way, such as by removing
   the copyright notice or references to the Internet Society or other
   Internet organizations, except as needed for the purpose of
   developing Internet standards in which case the procedures for

7.1. Simple GSS Context Initiator

```java
import org.ietf.jgss.*;

/**
 * This is a partial sketch for a simple client program that acts
 * as a GSS context initiator.  It illustrates how to use the Java
 * bindings for the GSS-API specified in
 * draft-ietf-cat-gssv2-javabind-03.txt.
 *
 * This code sketch assumes the existence of a GSS-API
 * implementation that supports the mechanism that it will need and
 * is present as a library package (org.ietf.jgss) either as part of
 * the standard JRE or in the CLASSPATH the application specifies.
 */

public class SimpleClient {

    private String serviceName; // name of peer (ie. server)
    private GSSCredential clientCred = null;
    private GSSContext context = null;
    private Oid mech; // underlying mechanism to use

    private GSSManager mgr = new GSSManager();

    ...
    ...

    private void clientActions() {

        initializeGSS();
```

```
    establishContext();
    doCommunication();
}

/**
 * Acquire credentials for the client.
 */
private void initializeGSS() {

    try {

        clientCred = mgr.createCredential(null /*default princ*/,
                GSSCredential.INDEFINITE /* max lifetime */,
                mech /* mechanism  to use */,
                GSSCredential.INITIATE_ONLY /* init context */);

        print("GSSCredential created for " +
              cred.getName().toString());
        print("Credential lifetime (sec)=" +
              cred.getRemainingLifetime());
    } catch (GSSException e) {
            print("GSS-API error in credential acquisition: "
                    + e.getMessage());
            ...
            ...
    }

    ...
    ...
}

/**
 * Does the security context establishment with the
 * server.
 */
private void establishContext() {

    byte[] inToken = new byte[0];
    byte[] outToken = null;

    try {

        GSSName peer = mgr.createName(serviceName,
                        GSSName.NT_HOSTBASED_SERVICE);

        context = mgr.createContext(peer, mech, gssCred,
                        GSSContext.INDEFINITE/*lifetime*/);
```

```
        // Will need to support confidentiality
        context.requestConf(true);

        while (!context.isEstablished()) {

            outToken = context.initSecContext(inToken, 0,
                                              inToken.length);

            if (outToken != null)
                writeGSSToken(outToken);

            if (!context.isEstablished())
                inToken = readGSSToken();
        }

        GSSName peer = context.getSrcName();
        print("Security context established with " + peer +
              " using underlying mechanism " + mech.toString());
    } catch (GSSException e) {
            print("GSS-API error during context establishment: "
                  + e.getMessage());
        ...
        ...
    }

    ...
    ...
}

/**
 * Sends some data to the server and reads back the
 * response.
 */
private void doCommunication()  {
        byte[] inToken = null;
        byte[] outToken = null;
        byte[] buffer;

        // Container for multiple input-output arguments to and
        // from the per-message routines (e.g., wrap/unwrap).
        MessageProp messgInfo = new MessageProp();

        try {

            /*
             * Now send some bytes to the server to be
             * processed. They will be integrity protected but
             * not encrypted for privacy.
```

```
     */

    buffer = readFromFile();

    // Set privacy to false and use the default QOP
    messgInfo.setPrivacy(false);

    outToken = context.wrap(buffer, 0, buffer.length,
                            messgInfo);

    writeGSSToken(outToken);

    /*
     * Now read the response from the server.
     */

    inToken = readGSSToken();
    buffer = context.unwrap(inToken, 0, inToken.length,
                            messgInfo);
    // All ok if no exception was thrown!

    GSSName peer = context.getSrcName();

    print("Message from "       + peer.toString()
          + " arrived.");
    print("Was it encrypted? "  +
          messgInfo.getPrivacy());
    print("Duplicate Token? "   +
          messgInfo.isDuplicateToken());
    print("Old Token? "         +
          messgInfo.isOldToken());
    print("Unsequenced Token? " +
          messgInfo.isUnseqToken());
    print("Gap Token? "         +
          messgInfo.isGapToken());

    ...
    ...

} catch (GSSException e) {
    print("GSS-API error in per-message calls: "
          + e.getMessage());
    ...
    ...
}

...
...
```

```
     } // end of doCommunication method

     ...
     ...

  } // end of class SimpleClient
```

7.2. **Simple GSS Context Acceptor**


```
  import org.ietf.jgss.*;

  /**
   * This is a partial sketch for a simple server program that acts
   * as a GSS context acceptor. It illustrates how to use the Java
   * bindings for the GSS-API specified in
   * draft-ietf-cat-gssv2-javabind-03.txt.
   *
   * This code sketch assumes the existence of a GSS-API
   * implementation that supports the mechanisms that it will need and
   * is present as a library package (org.ietf.jgss) either as part of
   * the standard JRE or in the CLASSPATH the application specifies.
   */

  import org.ietf.jgss.*;

  public class SimpleServer {

      private String serviceName;
      private GSSName name;
      private GSSCredential cred;

      private GSSManager mgr;

      ...
      ...

      /**
       * Wait for client connections, establish security contexts and
       * provide service.
       */
      private void loop() {

      ...
```

```
...

     mgr = new GSSManager();

     name = mgr.createName(serviceName,
               GSSName.NT_HOSTBASED_SERVICE);

     cred = mgr.createCredential(name,
               GSSCredential.INDEFINITE,
               null,
               GSSCredential.ACCEPT_ONLY);


     // Loop infinitely
     while (true) {

         Socket s = serverSock.acceptSecContext();

         // Start a new thread to serve this connection
         Thread serverThread = new ServerThread(s);
         serverThread.start();

     }
}

/**
 * Inner class ServerThread whose run() method provides the
 * secure service to a connection.
 */

private class ServerThread extends Thread {

...
...

     /**
      * Deals with the connection from one client. It also
      * handles all GSSException's thrown while talking to
      * this client.
      */
     public void run() {

         byte[] inToken = null;
         byte[] outToken = null;
         byte[] buffer;

         GSSName peer;
```

```
// Container for multiple input-output arguments to and
// from the per-message routines (ie. wrap/unwrap).
MessageProp supplInfo = new MessageProp();

GSSContext secContext = null;

try {


    // Now do the context establishment loop

    GSSContext context = mgr.createContext(cred);

    while (!context.isEstablished()) {

        inToken = readGSSToken();

        outToken = context.acceptSecContext(inToken, 0,
                                    inToken.length);

        if (outToken != null)
            writeGSSToken(outToken);

    }


    // SimpleServer wants confidentiality to be
    // available. Check for it.
    if (!context.getConfState()){
        ...
        ...
    }

    GSSName peer = context.getSrcName();
    Oid mech = context.getMech();
    print("Security context established with " +
            peer.toString() +
            " using underlying mechanism " +
            mech.toString() +
            " from Provider " +
            context.getProvider().getName());


    // Now read the bytes sent by the client to be
    // processed.
    inToken = readGSSToken();

    // Unwrap the message
```

```
        buffer = context.unwrap(inToken, 0, inToken.length,
                                supplInfo);
        // All ok if no exception was thrown!

        // Print other supplementary per-message status
        // information

        print("Message from " +
                peer.toString() + " arrived.");
        print("Was it encrypted? " +
                supplInfo.getPrivacy());
        print("Duplicate Token? " +
                supplInfo.isDuplicateToken());
        print("Old Token? "  + supplInfo.isOldToken());
        print("Unsequenced Token? " +
                supplInfo.isUnseqToken());
        print("Gap Token? "  + supplInfo.isGapToken());

        /*
         * Now process the bytes and send back an encrypted
         * response.
         */

        buffer = serverProcess(buffer);

        // Encipher it and send it across

        supplInfo.setPrivacy(true); // privacy requested
        supplInfo.setQOP(0); // default QOP
        outToken = context.wrap(buffer, 0, buffer.length,
                                supplInfo);
        writeGSSToken(outToken);

    } catch (GSSException e) {
        print("GSS-API Error: " + e.getMessage());
        // Alternatively, could call e.getMajorMessage()
        // and e.getMinorMessage()
        print("Abandoning security context.");

        ...
        ...

    }

    ...
    ...

} // end of run method in ServerThread
```

```
    } // end of inner class ServerThread

    ...
    ...

} // end of class SimpleServer
```

[8](#). Acknowledgments

   This proposed API leverages earlier work performed by the IETF's CAT
   WG as outlined in both [RFC 2078](#) and J. Wray's C-bindings draft for
   the GSS-API. Many conceptual definitions, implementation directions,
   and explanations have been included from the C-bindings draft.

   We would like to thank Mike Eisler, Lin Ling, Ram Marti, Michael
   Saltz and other members of Sun's development team for their helpful
   input, comments and suggestions.

   We would also like to thank Joe Salowey, and Michael Smith for many
   insightful ideas and suggestions that have contributed to this draft.

9. Bibliography

[GSSAPIv2]
J. Linn, "Generic Security Service Application Program Interface,
Version 2", RFC 2078, January 1997.

[GSSAPIv2-UPDATE]
J. Linn, "Generic Security Service Application Program Interface,
Version 2, Update 1", IETF work in progress, Internet Draft, July
1998.

[GSSAPI-Cbind]
J. Wray, "Generic Security Service API Version 2 : C-bindings", IETF
work in progress, Internet Draft, July 1998.

[KERBEROS_V5]
J. Linn, "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June
1996.

[SPKM]
C. Adams, "The Simple Public-Key GSS-API Mechanism", RFC 2025,
October 1996.

10. Author's Address

Address comments related to this memorandum to:

        <cat-ietf@mit.edu>

Jack Kabat
ValiCert, Inc.
1215 Terra Bella Avenue
Mountain View, CA
94043, USA

Phone: +1-650-567-5496
E-mail: jackk@valicert.com

Mayank Upadhyay
Sun Microsystems, Inc.
901 San Antonio Road, MS CUP02-102
Palo Alto, CA 94303

Phone: +1-408-517-5956
E-mail: mdu@eng.sun.com