                The Kerberos Network Authentication Service (V5)

Status of this Memo

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

Abstract

   This document gives an overview and specification of Version 5 of the
   protocol for the Kerberos network authentication system. Version 4,
   described elsewhere [1,2], is presently in production use at MIT's
   Project Athena, and at other Internet sites.

Overview

   Project Athena, Athena, Athena MUSE, Discuss, Hesiod, Kerberos,
   Moira, and Zephyr are trademarks of the Massachusetts Institute of
   Technology (MIT). No commercial use of these trademarks may be made
   without prior written permission of MIT.

   This RFC describes the concepts and model upon which the Kerberos
   network authentication system is based. It also specifies Version 5
   of the Kerberos protocol.

   The motivations, goals, assumptions, and rationale behind most design
   decisions are treated cursorily; for Version 4 they are fully
   described in the Kerberos portion of the Athena Technical Plan [1].
   The protocols are under review, and are not being submitted for
   consideration as an Internet standard at this time. Comments are
   encouraged. Requests for addition to an electronic mailing list for
   discussion of Kerberos, kerberos@MIT.EDU, may be addressed to
   kerberos-request@MIT.EDU. This mailing list is gatewayed onto the
   Usenet as the group comp.protocols.kerberos. Requests for further
   information, including documents and code availability, may be sent
   to info-kerberos@MIT.EDU.

Background

   The Kerberos model is based in part on Needham and Schroeder's
   trusted third-party authentication protocol [3] and on modifications
   suggested by Denning and Sacco [4]. The original design and
   implementation of Kerberos Versions 1 through 4 was the work of two
   former Project Athena staff members, Steve Miller of Digital
   Equipment Corporation and Clifford Neuman (now at the Information
   Sciences Institute of the University of Southern California), along
   with Jerome Saltzer, Technical Director of Project Athena, and
   Jeffrey Schiller, MIT Campus Network Manager. Many other members of
   Project Athena have also contributed to the work on Kerberos.
   Version 4 is publicly available, and has seen wide use across the
   Internet.

   Version 5 (described in this document) has evolved from Version 4
   based on new requirements and desires for features not available in
   Version 4. Details on the differences between Kerberos Versions 4
   and 5 can be found in [5].

Table of Contents

1. Introduction

      Kerberos provides a means of verifying the identities of principals,
      (e.g., a workstation user or a network server) on an open
      (unprotected) network. This is accomplished without relying on
      authentication by the host operating system, without basing trust on
      host addresses, without requiring physical security of all the hosts
      on the network, and under the assumption that packets traveling along
      the network can be read, modified, and inserted at will. (Note,
      however, that many applications use Kerberos' functions only upon the
      initiation of a stream-based network connection, and assume the
      absence of any "hijackers" who might subvert such a connection. Such
      use implicitly trusts the host addresses involved.) Kerberos
      performs authentication under these conditions as a trusted third-
      party authentication service by using conventional cryptography,
      i.e., shared secret key. (shared secret key - Secret and private are
      often used interchangeably in the literature. In our usage, it takes
      two (or more) to share a secret, thus a shared DES key is a secret
      key. Something is only private when no one but its owner knows it.
      Thus, in public key cryptosystems, one has a public and a private
      key.)

      The authentication process proceeds as follows: A client sends a
      request to the authentication server (AS) requesting "credentials"
      for a given server. The AS responds with these credentials,
      encrypted in the client's key. The credentials consist of 1) a
      "ticket" for the server and 2) a temporary encryption key (often
      called a "session key"). The client transmits the ticket (which
      contains the client's identity and a copy of the session key, all
      encrypted in the server's key) to the server. The session key (now
      shared by the client and server) is used to authenticate the client,
      and may optionally be used to authenticate the server. It may also
      be used to encrypt further communication between the two parties or
      to exchange a separate sub-session key to be used to encrypt further
      communication.

      The implementation consists of one or more authentication servers
      running on physically secure hosts. The authentication servers
      maintain a database of principals (i.e., users and servers) and their
      secret keys. Code libraries provide encryption and implement the
      Kerberos protocol. In order to add authentication to its

transactions, a typical network application adds one or two calls to
the Kerberos library, which results in the transmission of the
necessary messages to achieve authentication.

The Kerberos protocol consists of several sub-protocols (or
exchanges). There are two methods by which a client can ask a
Kerberos server for credentials. In the first approach, the client
sends a cleartext request for a ticket for the desired server to the
AS. The reply is sent encrypted in the client's secret key. Usually
this request is for a ticket-granting ticket (TGT) which can later be
used with the ticket-granting server (TGS). In the second method,
the client sends a request to the TGS. The client sends the TGT to
the TGS in the same manner as if it were contacting any other
application server which requires Kerberos credentials. The reply is
encrypted in the session key from the TGT.

Once obtained, credentials may be used to verify the identity of the
principals in a transaction, to ensure the integrity of messages
exchanged between them, or to preserve privacy of the messages. The
application is free to choose whatever protection may be necessary.

To verify the identities of the principals in a transaction, the
client transmits the ticket to the server. Since the ticket is sent
"in the clear" (parts of it are encrypted, but this encryption
doesn't thwart replay) and might be intercepted and reused by an
attacker, additional information is sent to prove that the message
was originated by the principal to whom the ticket was issued. This
information (called the authenticator) is encrypted in the session
key, and includes a timestamp. The timestamp proves that the message
was recently generated and is not a replay. Encrypting the
authenticator in the session key proves that it was generated by a
party possessing the session key. Since no one except the requesting
principal and the server know the session key (it is never sent over
the network in the clear) this guarantees the identity of the client.

The integrity of the messages exchanged between principals can also
be guaranteed using the session key (passed in the ticket and
contained in the credentials). This approach provides detection of
both replay attacks and message stream modification attacks. It is
accomplished by generating and transmitting a collision-proof
checksum (elsewhere called a hash or digest function) of the client's
message, keyed with the session key. Privacy and integrity of the
messages exchanged between principals can be secured by encrypting
the data to be passed using the session key passed in the ticket, and
contained in the credentials.

The authentication exchanges mentioned above require read-only access
to the Kerberos database. Sometimes, however, the entries in the

database must be modified, such as when adding new principals or
changing a principal's key. This is done using a protocol between a
client and a third Kerberos server, the Kerberos Administration
Server (KADM). The administration protocol is not described in this
document. There is also a protocol for maintaining multiple copies of
the Kerberos database, but this can be considered an implementation
detail and may vary to support different database technologies.

1.1. Cross-Realm Operation

The Kerberos protocol is designed to operate across organizational
boundaries. A client in one organization can be authenticated to a
server in another. Each organization wishing to run a Kerberos
server establishes its own "realm". The name of the realm in which a
client is registered is part of the client's name, and can be used by
the end-service to decide whether to honor a request.

By establishing "inter-realm" keys, the administrators of two realms
can allow a client authenticated in the local realm to use its
authentication remotely (Of course, with appropriate permission the
client could arrange registration of a separately-named principal in
a remote realm, and engage in normal exchanges with that realm's
services. However, for even small numbers of clients this becomes
cumbersome, and more automatic methods as described here are
necessary). The exchange of inter-realm keys (a separate key may be
used for each direction) registers the ticket-granting service of
each realm as a principal in the other realm. A client is then able
to obtain a ticket-granting ticket for the remote realm's ticket-
granting service from its local realm. When that ticket-granting
ticket is used, the remote ticket-granting service uses the inter-
realm key (which usually differs from its own normal TGS key) to
decrypt the ticket-granting ticket, and is thus certain that it was
issued by the client's own TGS. Tickets issued by the remote ticket-
granting service will indicate to the end-service that the client was
authenticated from another realm.

A realm is said to communicate with another realm if the two realms
share an inter-realm key, or if the local realm shares an inter-realm
key with an intermediate realm that communicates with the remote
realm. An authentication path is the sequence of intermediate realms
that are transited in communicating from one realm to another.

Realms are typically organized hierarchically. Each realm shares a
key with its parent and a different key with each child. If an
inter-realm key is not directly shared by two realms, the
hierarchical organization allows an authentication path to be easily
constructed. If a hierarchical organization is not used, it may be
necessary to consult some database in order to construct an

authentication path between realms.

Although realms are typically hierarchical, intermediate realms may
be bypassed to achieve cross-realm authentication through alternate
authentication paths (these might be established to make
communication between two realms more efficient). It is important
for the end-service to know which realms were transited when deciding
how much faith to place in the authentication process. To facilitate
this decision, a field in each ticket contains the names of the
realms that were involved in authenticating the client.

1.2. Environmental assumptions

Kerberos imposes a few assumptions on the environment in which it can
properly function:

+   "Denial of service" attacks are not solved with Kerberos. There
     are places in these protocols where an intruder intruder can
     prevent an application from participating in the proper
     authentication steps. Detection and solution of such attacks
     (some of which can appear to be not-uncommon "normal" failure
     modes for the system) is usually best left to the human
     administrators and users.

+   Principals must keep their secret keys secret. If an intruder
     somehow steals a principal's key, it will be able to masquerade
     as that principal or impersonate any server to the legitimate
     principal.

+   "Password guessing" attacks are not solved by Kerberos. If a
     user chooses a poor password, it is possible for an attacker to
     successfully mount an offline dictionary attack by repeatedly
     attempting to decrypt, with successive entries from a
     dictionary, messages obtained which are encrypted under a key
     derived from the user's password.

+   Each host on the network must have a clock which is "loosely
     synchronized" to the time of the other hosts; this
     synchronization is used to reduce the bookkeeping needs of
     application servers when they do replay detection. The degree
     of "looseness" can be configured on a per-server basis. If the
     clocks are synchronized over the network, the clock
     synchronization protocol must itself be secured from network
     attackers.

+   Principal identifiers are not recycled on a short-term basis. A
     typical mode of access control will use access control lists
     (ACLs) to grant permissions to particular principals. If a

        stale ACL entry remains for a deleted principal and the
        principal identifier is reused, the new principal will inherit
        rights specified in the stale ACL entry. By not re-using
        principal identifiers, the danger of inadvertent access is
        removed.

1.3. Glossary of terms

    Below is a list of terms used throughout this document.

    Authentication   Verifying the claimed identity of a
                          principal.

    Authentication header A record containing a Ticket and an
                          Authenticator to be presented to a
                          server as part of the authentication
                          process.

    Authentication path   A sequence of intermediate realms transited
                          in the authentication process when
                          communicating from one realm to another.

    Authenticator   A record containing information that can
                          be shown to have been recently generated
                          using the session key known only by the
                          client and server.

    Authorization   The process of determining whether a
                          client may use a service, which objects
                          the client is allowed to access, and the
                          type of access allowed for each.

    Capability   A token that grants the bearer permission
                          to access an object or service. In
                          Kerberos, this might be a ticket whose
                          use is restricted by the contents of the
                          authorization data field, but which
                          lists no network addresses, together
                          with the session key necessary to use
                          the ticket.

Ciphertext   The output of an encryption function.
                        Encryption transforms plaintext into
                        ciphertext.

Client   A process that makes use of a network
                        service on behalf of a user. Note that
                        in some cases a Server may itself be a
                        client of some other server (e.g., a
                        print server may be a client of a file
                        server).

Credentials   A ticket plus the secret session key
                        necessary to successfully use that
                        ticket in an authentication exchange.

KDC   Key Distribution Center, a network service
                        that supplies tickets and temporary
                        session keys; or an instance of that
                        service or the host on which it runs.
                        The KDC services both initial ticket and
                        ticket-granting ticket requests. The
                        initial ticket portion is sometimes
                        referred to as the Authentication Server
                        (or service). The ticket-granting
                        ticket portion is sometimes referred to
                        as the ticket-granting server (or service).

Kerberos   Aside from the 3-headed dog guarding
                        Hades, the name given to Project
                        Athena's authentication service, the
                        protocol used by that service, or the
                        code used to implement the authentication
                        service.

Plaintext   The input to an encryption function or
                        the output of a decryption function.
                        Decryption transforms ciphertext into
                        plaintext.

Principal   A uniquely named client or server
                        instance that participates in a network
                        communication.

Principal identifier The name used to uniquely identify each
                     different principal.

Seal   To encipher a record containing several
                     fields in such a way that the fields
                     cannot be individually replaced without
                     either knowledge of the encryption key
                     or leaving evidence of tampering.

Secret key   An encryption key shared by a principal
                     and the KDC, distributed outside the
                     bounds of the system, with a long lifetime.
                     In the case of a human user's
                     principal, the secret key is derived
                     from a password.

Server   A particular Principal which provides a
                     resource to network clients.

Service   A resource provided to network clients;
                     often provided by more than one server
                     (for example, remote file service).

Session key   A temporary encryption key used between
                     two principals, with a lifetime limited
                     to the duration of a single login "session".

Sub-session key   A temporary encryption key used between
                     two principals, selected and exchanged
                     by the principals using the session key,
                     and with a lifetime limited to the duration
                     of a single association.

Ticket   A record that helps a client authenticate
                     itself to a server; it contains the
                     client's identity, a session key, a
                     timestamp, and other information, all
                     sealed using the server's secret key.
                     It only serves to authenticate a client
                     when presented along with a fresh
                     Authenticator.

2. Ticket flag uses and requests

   Each Kerberos ticket contains a set of flags which are used to
   indicate various attributes of that ticket. Most flags may be
   requested by a client when the ticket is obtained; some are
   automatically turned on and off by a Kerberos server as required.
   The following sections explain what the various flags mean, and gives
   examples of reasons to use such a flag.

2.1. Initial and pre-authenticated tickets

   The INITIAL flag indicates that a ticket was issued using the AS
   protocol and not issued based on a ticket-granting ticket.
   Application servers that want to require the knowledge of a client's
   secret key (e.g., a passwordchanging program) can insist that this
   flag be set in any tickets they accept, and thus be assured that the
   client's key was recently presented to the application client.

   The PRE-AUTHENT and HW-AUTHENT flags provide addition information
   about the initial authentication, regardless of whether the current
   ticket was issued directly (in which case INITIAL will also be set)
   or issued on the basis of a ticket-granting ticket (in which case the
   INITIAL flag is clear, but the PRE-AUTHENT and HW-AUTHENT flags are
   carried forward from the ticket-granting ticket).

2.2. Invalid tickets

   The INVALID flag indicates that a ticket is invalid. Application
   servers must reject tickets which have this flag set. A postdated
   ticket will usually be issued in this form. Invalid tickets must be
   validated by the KDC before use, by presenting them to the KDC in a
   TGS request with the VALIDATE option specified. The KDC will only
   validate tickets after their starttime has passed. The validation is
   required so that postdated tickets which have been stolen before
   their starttime can be rendered permanently invalid (through a hot-
   list mechanism).

2.3. Renewable tickets

   Applications may desire to hold tickets which can be valid for long
   periods of time. However, this can expose their credentials to
   potential theft for equally long periods, and those stolen
   credentials would be valid until the expiration time of the
   ticket(s). Simply using shortlived tickets and obtaining new ones
   periodically would require the client to have long-term access to its
   secret key, an even greater risk. Renewable tickets can be used to
   mitigate the consequences of theft. Renewable tickets have two
   "expiration times": the first is when the current instance of the

ticket expires, and the second is the latest permissible value for an
individual expiration time. An application client must periodically
(i.e., before it expires) present a renewable ticket to the KDC, with
the RENEW option set in the KDC request. The KDC will issue a new
ticket with a new session key and a later expiration time. All other
fields of the ticket are left unmodified by the renewal process.
When the latest permissible expiration time arrives, the ticket
expires permanently. At each renewal, the KDC may consult a hot-list
to determine if the ticket had been reported stolen since its last
renewal; it will refuse to renew such stolen tickets, and thus the
usable lifetime of stolen tickets is reduced.

The RENEWABLE flag in a ticket is normally only interpreted by the
ticket-granting service (discussed below in section 3.3). It can
usually be ignored by application servers. However, some
particularly careful application servers may wish to disallow
renewable tickets.

If a renewable ticket is not renewed by its expiration time, the KDC
will not renew the ticket. The RENEWABLE flag is reset by default,
but a client may request it be set by setting the RENEWABLE option
in the KRB_AS_REQ message. If it is set, then the renew-till field
in the ticket contains the time after which the ticket may not be
renewed.

2.4. Postdated tickets

Applications may occasionally need to obtain tickets for use much
later, e.g., a batch submission system would need tickets to be valid
at the time the batch job is serviced. However, it is dangerous to
hold valid tickets in a batch queue, since they will be on-line
longer and more prone to theft. Postdated tickets provide a way to
obtain these tickets from the KDC at job submission time, but to
leave them "dormant" until they are activated and validated by a
further request of the KDC. If a ticket theft were reported in the
interim, the KDC would refuse to validate the ticket, and the thief
would be foiled.

The MAY-POSTDATE flag in a ticket is normally only interpreted by the
ticket-granting service. It can be ignored by application servers.
This flag must be set in a ticket-granting ticket in order to issue a
postdated ticket based on the presented ticket. It is reset by
default; it may be requested by a client by setting the ALLOW-
POSTDATE option in the KRB_AS_REQ message. This flag does not allow
a client to obtain a postdated ticket-granting ticket; postdated
ticket-granting tickets can only by obtained by requesting the
postdating in the KRB_AS_REQ message. The life (endtime-starttime)
of a postdated ticket will be the remaining life of the ticket-

granting ticket at the time of the request, unless the RENEWABLE
option is also set, in which case it can be the full life (endtime-
starttime) of the ticket-granting ticket. The KDC may limit how far
in the future a ticket may be postdated.

The POSTDATED flag indicates that a ticket has been postdated. The
application server can check the authtime field in the ticket to see
when the original authentication occurred. Some services may choose
to reject postdated tickets, or they may only accept them within a
certain period after the original authentication. When the KDC issues
a POSTDATED ticket, it will also be marked as INVALID, so that the
application client must present the ticket to the KDC to be validated
before use.

2.5. Proxiable and proxy tickets

At times it may be necessary for a principal to allow a service to
perform an operation on its behalf. The service must be able to take
on the identity of the client, but only for a particular purpose. A
principal can allow a service to take on the principal's identity for
a particular purpose by granting it a proxy.

The PROXIABLE flag in a ticket is normally only interpreted by the
ticket-granting service. It can be ignored by application servers.
When set, this flag tells the ticket-granting server that it is OK to
issue a new ticket (but not a ticket-granting ticket) with a
different network address based on this ticket. This flag is set by
default.

This flag allows a client to pass a proxy to a server to perform a
remote request on its behalf, e.g., a print service client can give
the print server a proxy to access the client's files on a particular
file server in order to satisfy a print request.

In order to complicate the use of stolen credentials, Kerberos
tickets are usually valid from only those network addresses
specifically included in the ticket (It is permissible to request or
issue tickets with no network addresses specified, but we do not
recommend it). For this reason, a client wishing to grant a proxy
must request a new ticket valid for the network address of the
service to be granted the proxy.

The PROXY flag is set in a ticket by the TGS when it issues a
proxy ticket. Application servers may check this flag and require
additional authentication from the agent presenting the proxy in
order to provide an audit trail.

2.6. Forwardable tickets

   Authentication forwarding is an instance of the proxy case where the
   service is granted complete use of the client's identity. An example
   where it might be used is when a user logs in to a remote system and
   wants authentication to work from that system as if the login were
   local.

   The FORWARDABLE flag in a ticket is normally only interpreted by the
   ticket-granting service. It can be ignored by application servers.
   The FORWARDABLE flag has an interpretation similar to that of the
   PROXIABLE flag, except ticket-granting tickets may also be issued
   with different network addresses. This flag is reset by default, but
   users may request that it be set by setting the FORWARDABLE option in
   the AS request when they request their initial ticket-granting
   ticket.

   This flag allows for authentication forwarding without requiring the
   user to enter a password again. If the flag is not set, then
   authentication forwarding is not permitted, but the same end result
   can still be achieved if the user engages in the AS exchange with the
   requested network addresses and supplies a password.

   The FORWARDED flag is set by the TGS when a client presents a ticket
   with the FORWARDABLE flag set and requests it be set by specifying
   the FORWARDED KDC option and supplying a set of addresses for the new
   ticket. It is also set in all tickets issued based on tickets with
   the FORWARDED flag set. Application servers may wish to process
   FORWARDED tickets differently than non-FORWARDED tickets.

2.7. Other KDC options

   There are two additional options which may be set in a client's
   request of the KDC. The RENEWABLE-OK option indicates that the
   client will accept a renewable ticket if a ticket with the requested
   life cannot otherwise be provided. If a ticket with the requested
   life cannot be provided, then the KDC may issue a renewable ticket
   with a renew-till equal to the the requested endtime. The value of
   the renew-till field may still be adjusted by site-determined limits
   or limits imposed by the individual principal or server.

   The ENC-TKT-IN-SKEY option is honored only by the ticket-granting
   service. It indicates that the to-be-issued ticket for the end
   server is to be encrypted in the session key from the additional
   ticket-granting ticket provided with the request. See section 3.3.3
   for specific details.

3. Message Exchanges

   The following sections describe the interactions between network
   clients and servers and the messages involved in those exchanges.

3.1. The Authentication Service Exchange

                         Summary

         Message direction   Message type   Section
         1. Client to Kerberos   KRB_AS_REQ   5.4.1
         2. Kerberos to client   KRB_AS_REP or   5.4.2
                                    KRB_ERROR   5.9.1

   The Authentication Service (AS) Exchange between the client and the
   Kerberos Authentication Server is usually initiated by a client when
   it wishes to obtain authentication credentials for a given server but
   currently holds no credentials. The client's secret key is used for
   encryption and decryption. This exchange is typically used at the
   initiation of a login session, to obtain credentials for a Ticket-
   Granting Server, which will subsequently be used to obtain
   credentials for other servers (see section 3.3) without requiring
   further use of the client's secret key. This exchange is also used
   to request credentials for services which must not be mediated
   through the Ticket-Granting Service, but rather require a principal's
   secret key, such as the password-changing service. (The password-
   changing request must not be honored unless the requester can provide
   the old password (the user's current secret key). Otherwise, it
   would be possible for someone to walk up to an unattended session and
   change another user's password.) This exchange does not by itself
   provide any assurance of the the identity of the user. (To
   authenticate a user logging on to a local system, the credentials
   obtained in the AS exchange may first be used in a TGS exchange to
   obtain credentials for a local server. Those credentials must then
   be verified by the local server through successful completion of the
   Client/Server exchange.)

   The exchange consists of two messages: KRB_AS_REQ from the client to
   Kerberos, and KRB_AS_REP or KRB_ERROR in reply. The formats for these
   messages are described in sections 5.4.1, 5.4.2, and 5.9.1.

   In the request, the client sends (in cleartext) its own identity and
   the identity of the server for which it is requesting credentials.
   The response, KRB_AS_REP, contains a ticket for the client to present
   to the server, and a session key that will be shared by the client
   and the server. The session key and additional information are
   encrypted in the client's secret key. The KRB_AS_REP message
   contains information which can be used to detect replays, and to

associate it with the message to which it replies. Various errors
can occur; these are indicated by an error response (KRB_ERROR)
instead of the KRB_AS_REP response. The error message is not
encrypted. The KRB_ERROR message also contains information which can
be used to associate it with the message to which it replies. The
lack of encryption in the KRB_ERROR message precludes the ability to
detect replays or fabrications of such messages.

In the normal case the authentication server does not know whether
the client is actually the principal named in the request. It simply
sends a reply without knowing or caring whether they are the same.
This is acceptable because nobody but the principal whose identity
was given in the request will be able to use the reply. Its critical
information is encrypted in that principal's key. The initial
request supports an optional field that can be used to pass
additional information that might be needed for the initial exchange.
This field may be used for preauthentication if desired, but the
mechanism is not currently specified.

3.1.1. Generation of KRB_AS_REQ message

The client may specify a number of options in the initial request.
Among these options are whether preauthentication is to be performed;
whether the requested ticket is to be renewable, proxiable, or
forwardable; whether it should be postdated or allow postdating of
derivative tickets; and whether a renewable ticket will be accepted
in lieu of a non-renewable ticket if the requested ticket expiration
date cannot be satisfied by a nonrenewable ticket (due to
configuration constraints; see section 4). See section A.1 for
pseudocode.

The client prepares the KRB_AS_REQ message and sends it to the KDC.

3.1.2. Receipt of KRB_AS_REQ message

If all goes well, processing the KRB_AS_REQ message will result in
the creation of a ticket for the client to present to the server.
The format for the ticket is described in section 5.3.1. The
contents of the ticket are determined as follows.

3.1.3. Generation of KRB_AS_REP message

The authentication server looks up the client and server principals
named in the KRB_AS_REQ in its database, extracting their respective
keys. If required, the server pre-authenticates the request, and if
the pre-authentication check fails, an error message with the code
KDC_ERR_PREAUTH_FAILED is returned. If the server cannot accommodate
the requested encryption type, an error message with code

KDC_ERR_ETYPE_NOSUPP is returned. Otherwise it generates a "random"
session key ("Random" means that, among other things, it should be
impossible to guess the next session key based on knowledge of past
session keys. This can only be achieved in a pseudo-random number
generator if it is based on cryptographic principles. It would be
more desirable to use a truly random number generator, such as one
based on measurements of random physical phenomena.).

If the requested start time is absent or indicates a time in the
past, then the start time of the ticket is set to the authentication
server's current time. If it indicates a time in the future, but the
POSTDATED option has not been specified, then the error
KDC_ERR_CANNOT_POSTDATE is returned. Otherwise the requested start
time is checked against the policy of the local realm (the
administrator might decide to prohibit certain types or ranges of
postdated tickets), and if acceptable, the ticket's start time is set
as requested and the INVALID flag is set in the new ticket. The
postdated ticket must be validated before use by presenting it to the
KDC after the start time has been reached.

The expiration time of the ticket will be set to the minimum of the
following:

+The expiration time (endtime) requested in the KRB_AS_REQ
 message.

+The ticket's start time plus the maximum allowable lifetime
 associated with the client principal (the authentication
 server's database includes a maximum ticket lifetime field
 in each principal's record; see section 4).

+The ticket's start time plus the maximum allowable lifetime
 associated with the server principal.

+The ticket's start time plus the maximum lifetime set by
 the policy of the local realm.

If the requested expiration time minus the start time (as determined
above) is less than a site-determined minimum lifetime, an error
message with code KDC_ERR_NEVER_VALID is returned. If the requested
expiration time for the ticket exceeds what was determined as above,
and if the "RENEWABLE-OK" option was requested, then the "RENEWABLE"
flag is set in the new ticket, and the renew-till value is set as if
the "RENEWABLE" option were requested (the field and option names are
described fully in section 5.4.1). If the RENEWABLE option has been
requested or if the RENEWABLE-OK option has been set and a renewable
ticket is to be issued, then the renew-till field is set to the
minimum of:

+Its requested value.

+The start time of the ticket plus the minimum of the two
 maximum renewable lifetimes associated with the principals'
 database entries.

+The start time of the ticket plus the maximum renewable
 lifetime set by the policy of the local realm.

The flags field of the new ticket will have the following options set
if they have been requested and if the policy of the local realm
allows: FORWARDABLE, MAY-POSTDATE, POSTDATED, PROXIABLE, RENEWABLE.
If the new ticket is postdated (the start time is in the future), its
INVALID flag will also be set.

If all of the above succeed, the server formats a KRB_AS_REP message
(see section 5.4.2), copying the addresses in the request into the
caddr of the response, placing any required pre-authentication data
into the padata of the response, and encrypts the ciphertext part in
the client's key using the requested encryption method, and sends it
to the client. See section A.2 for pseudocode.

3.1.4. Generation of KRB_ERROR message

Several errors can occur, and the Authentication Server responds by
returning an error message, KRB_ERROR, to the client, with the
error-code and e-text fields set to appropriate values. The error
message contents and details are described in Section 5.9.1.

3.1.5. Receipt of KRB_AS_REP message

If the reply message type is KRB_AS_REP, then the client verifies
that the cname and crealm fields in the cleartext portion of the
reply match what it requested. If any padata fields are present,
they may be used to derive the proper secret key to decrypt the
message. The client decrypts the encrypted part of the response
using its secret key, verifies that the nonce in the encrypted part
matches the nonce it supplied in its request (to detect replays). It
also verifies that the sname and srealm in the response match those
in the request, and that the host address field is also correct. It
then stores the ticket, session key, start and expiration times, and
other information for later use. The key-expiration field from the
encrypted part of the response may be checked to notify the user of
impending key expiration (the client program could then suggest
remedial action, such as a password change). See section A.3 for
pseudocode.

Proper decryption of the KRB_AS_REP message is not sufficient to

verify the identity of the user; the user and an attacker could
cooperate to generate a KRB_AS_REP format message which decrypts
properly but is not from the proper KDC. If the host wishes to
verify the identity of the user, it must require the user to present
application credentials which can be verified using a securely-stored
secret key. If those credentials can be verified, then the identity
of the user can be assured.

3.1.6. Receipt of KRB_ERROR message

If the reply message type is KRB_ERROR, then the client interprets it
as an error and performs whatever application-specific tasks are
necessary to recover.

3.2. The Client/Server Authentication Exchange

                    Summary

Message direction   Message type   Section
Client to Application server   KRB_AP_REQ   5.5.1
[optional] Application server to client   KRB_AP_REP or   5.5.2
                                             KRB_ERROR   5.9.1

The client/server authentication (CS) exchange is used by network
applications to authenticate the client to the server and vice versa.
The client must have already acquired credentials for the server
using the AS or TGS exchange.

3.2.1. The KRB_AP_REQ message

The KRB_AP_REQ contains authentication information which should be
part of the first message in an authenticated transaction. It
contains a ticket, an authenticator, and some additional bookkeeping
information (see section 5.5.1 for the exact format). The ticket by
itself is insufficient to authenticate a client, since tickets are
passed across the network in cleartext(Tickets contain both an
encrypted and unencrypted portion, so cleartext here refers to the
entire unit, which can be copied from one message and replayed in
another without any cryptographic skill.), so the authenticator is
used to prevent invalid replay of tickets by proving to the server
that the client knows the session key of the ticket and thus is
entitled to use it. The KRB_AP_REQ message is referred to elsewhere
as the "authentication header."

3.2.2. Generation of a KRB_AP_REQ message

When a client wishes to initiate authentication to a server, it
obtains (either through a credentials cache, the AS exchange, or the

TGS exchange) a ticket and session key for the desired service. The
client may re-use any tickets it holds until they expire. The client
then constructs a new Authenticator from the the system time, its
name, and optionally an application specific checksum, an initial
sequence number to be used in KRB_SAFE or KRB_PRIV messages, and/or a
session subkey to be used in negotiations for a session key unique to
this particular session. Authenticators may not be re-used and will
be rejected if replayed to a server (Note that this can make
applications based on unreliable transports difficult to code
correctly, if the transport might deliver duplicated messages. In
such cases, a new authenticator must be generated for each retry.).
If a sequence number is to be included, it should be randomly chosen
so that even after many messages have been exchanged it is not likely
to collide with other sequence numbers in use.

The client may indicate a requirement of mutual authentication or the
use of a session-key based ticket by setting the appropriate flag(s)
in the ap-options field of the message.

The Authenticator is encrypted in the session key and combined with
the ticket to form the KRB_AP_REQ message which is then sent to the
end server along with any additional application-specific
information. See section A.9 for pseudocode.

3.2.3. Receipt of KRB_AP_REQ message

Authentication is based on the server's current time of day (clocks
must be loosely synchronized), the authenticator, and the ticket.
Several errors are possible. If an error occurs, the server is
expected to reply to the client with a KRB_ERROR message. This
message may be encapsulated in the application protocol if its "raw"
form is not acceptable to the protocol. The format of error messages
is described in section 5.9.1.

The algorithm for verifying authentication information is as follows.
If the message type is not KRB_AP_REQ, the server returns the
KRB_AP_ERR_MSG_TYPE error. If the key version indicated by the Ticket
in the KRB_AP_REQ is not one the server can use (e.g., it indicates
an old key, and the server no longer possesses a copy of the old
key), the KRB_AP_ERR_BADKEYVER error is returned. If the USE-
SESSION-KEY flag is set in the ap-options field, it indicates to the
server that the ticket is encrypted in the session key from the
server's ticket-granting ticket rather than its secret key (This is
used for user-to-user authentication as described in [6]). Since it
is possible for the server to be registered in multiple realms, with
different keys in each, the srealm field in the unencrypted portion
of the ticket in the KRB_AP_REQ is used to specify which secret key
the server should use to decrypt that ticket. The KRB_AP_ERR_NOKEY

error code is returned if the server doesn't have the proper key to
decipher the ticket.

The ticket is decrypted using the version of the server's key
specified by the ticket. If the decryption routines detect a
modification of the ticket (each encryption system must provide
safeguards to detect modified ciphertext; see section 6), the
KRB_AP_ERR_BAD_INTEGRITY error is returned (chances are good that
different keys were used to encrypt and decrypt).

The authenticator is decrypted using the session key extracted from
the decrypted ticket. If decryption shows it to have been modified,
the KRB_AP_ERR_BAD_INTEGRITY error is returned. The name and realm
of the client from the ticket are compared against the same fields in
the authenticator. If they don't match, the KRB_AP_ERR_BADMATCH
error is returned (they might not match, for example, if the wrong
session key was used to encrypt the authenticator). The addresses in
the ticket (if any) are then searched for an address matching the
operating-system reported address of the client. If no match is
found or the server insists on ticket addresses but none are present
in the ticket, the KRB_AP_ERR_BADADDR error is returned.

If the local (server) time and the client time in the authenticator
differ by more than the allowable clock skew (e.g., 5 minutes), the
KRB_AP_ERR_SKEW error is returned. If the server name, along with
the client name, time and microsecond fields from the Authenticator
match any recently-seen such tuples, the KRB_AP_ERR_REPEAT error is
returned (Note that the rejection here is restricted to
authenticators from the same principal to the same server. Other
client principals communicating with the same server principal should
not be have their authenticators rejected if the time and microsecond
fields happen to match some other client's authenticator.). The
server must remember any authenticator presented within the allowable
clock skew, so that a replay attempt is guaranteed to fail. If a
server loses track of any authenticator presented within the
allowable clock skew, it must reject all requests until the clock
skew interval has passed. This assures that any lost or re-played
authenticators will fall outside the allowable clock skew and can no
longer be successfully replayed (If this is not done, an attacker
could conceivably record the ticket and authenticator sent over the
network to a server, then disable the client's host, pose as the
disabled host, and replay the ticket and authenticator to subvert the
authentication.). If a sequence number is provided in the
authenticator, the server saves it for later use in processing
KRB_SAFE and/or KRB_PRIV messages. If a subkey is present, the
server either saves it for later use or uses it to help generate its
own choice for a subkey to be returned in a KRB_AP_REP message.

The server computes the age of the ticket: local (server) time minus
the start time inside the Ticket. If the start time is later than
the current time by more than the allowable clock skew or if the
INVALID flag is set in the ticket, the KRB_AP_ERR_TKT_NYV error is
returned. Otherwise, if the current time is later than end time by
more than the allowable clock skew, the KRB_AP_ERR_TKT_EXPIRED error
is returned.

If all these checks succeed without an error, the server is assured
that the client possesses the credentials of the principal named in
the ticket and thus, the client has been authenticated to the server.
See section A.10 for pseudocode.

3.2.4. Generation of a KRB_AP_REP message

Typically, a client's request will include both the authentication
information and its initial request in the same message, and the
server need not explicitly reply to the KRB_AP_REQ. However, if
mutual authentication (not only authenticating the client to the
server, but also the server to the client) is being performed, the
KRB_AP_REQ message will have MUTUAL-REQUIRED set in its ap-options
field, and a KRB_AP_REP message is required in response. As with the
error message, this message may be encapsulated in the application
protocol if its "raw" form is not acceptable to the application's
protocol. The timestamp and microsecond field used in the reply must
be the client's timestamp and microsecond field (as provided in the
authenticator). [Note: In the Kerberos version 4 protocol, the
timestamp in the reply was the client's timestamp plus one. This is
not necessary in version 5 because version 5 messages are formatted
in such a way that it is not possible to create the reply by
judicious message surgery (even in encrypted form) without knowledge
of the appropriate encryption keys.] If a sequence number is to be
included, it should be randomly chosen as described above for the
authenticator. A subkey may be included if the server desires to
negotiate a different subkey. The KRB_AP_REP message is encrypted in
the session key extracted from the ticket. See section A.11 for
pseudocode.

3.2.5. Receipt of KRB_AP_REP message

If a KRB_AP_REP message is returned, the client uses the session key
from the credentials obtained for the server (Note that for
encrypting the KRB_AP_REP message, the sub-session key is not used,
even if present in the Authenticator.) to decrypt the message, and
verifies that the timestamp and microsecond fields match those in the
Authenticator it sent to the server. If they match, then the client
is assured that the server is genuine. The sequence number and subkey
(if present) are retained for later use. See section A.12 for

pseudocode.

3.2.6. Using the encryption key

After the KRB_AP_REQ/KRB_AP_REP exchange has occurred, the client and
server share an encryption key which can be used by the application.
The "true session key" to be used for KRB_PRIV, KRB_SAFE, or other
application-specific uses may be chosen by the application based on
the subkeys in the KRB_AP_REP message and the authenticator
(Implementations of the protocol may wish to provide routines to
choose subkeys based on session keys and random numbers and to
orchestrate a negotiated key to be returned in the KRB_AP_REP
message.). In some cases, the use of this session key will be
implicit in the protocol; in others the method of use must be chosen
from a several alternatives. We leave the protocol negotiations of
how to use the key (e.g., selecting an encryption or checksum type)
to the application programmer; the Kerberos protocol does not
constrain the implementation options.

With both the one-way and mutual authentication exchanges, the peers
should take care not to send sensitive information to each other
without proper assurances. In particular, applications that require
privacy or integrity should use the KRB_AP_REP or KRB_ERROR responses
from the server to client to assure both client and server of their
peer's identity. If an application protocol requires privacy of its
messages, it can use the KRB_PRIV message (section 3.5). The KRB_SAFE
message (section 3.4) can be used to assure integrity.

3.3. The Ticket-Granting Service (TGS) Exchange

                            Summary

Message direction	Message type	Section
1. Client to Kerberos	KRB_TGS_REQ	5.4.1
2. Kerberos to client	KRB_TGS_REP or	5.4.2
	KRB_ERROR	5.9.1

The TGS exchange between a client and the Kerberos Ticket-Granting
Server is initiated by a client when it wishes to obtain
authentication credentials for a given server (which might be
registered in a remote realm), when it wishes to renew or validate an
existing ticket, or when it wishes to obtain a proxy ticket. In the
first case, the client must already have acquired a ticket for the
Ticket-Granting Service using the AS exchange (the ticket-granting
ticket is usually obtained when a client initially authenticates to
the system, such as when a user logs in). The message format for the
TGS exchange is almost identical to that for the AS exchange. The
primary difference is that encryption and decryption in the TGS

exchange does not take place under the client's key. Instead, the
session key from the ticket-granting ticket or renewable ticket, or
sub-session key from an Authenticator is used. As is the case for
all application servers, expired tickets are not accepted by the TGS,
so once a renewable or ticket-granting ticket expires, the client
must use a separate exchange to obtain valid tickets.

The TGS exchange consists of two messages: A request (KRB_TGS_REQ)
from the client to the Kerberos Ticket-Granting Server, and a reply
(KRB_TGS_REP or KRB_ERROR). The KRB_TGS_REQ message includes
information authenticating the client plus a request for credentials.
The authentication information consists of the authentication header
(KRB_AP_REQ) which includes the client's previously obtained ticket-
granting, renewable, or invalid ticket. In the ticket-granting
ticket and proxy cases, the request may include one or more of: a
list of network addresses, a collection of typed authorization data
to be sealed in the ticket for authorization use by the application
server, or additional tickets (the use of which are described later).
The TGS reply (KRB_TGS_REP) contains the requested credentials,
encrypted in the session key from the ticket-granting ticket or
renewable ticket, or if present, in the subsession key from the
Authenticator (part of the authentication header). The KRB_ERROR
message contains an error code and text explaining what went wrong.
The KRB_ERROR message is not encrypted. The KRB_TGS_REP message
contains information which can be used to detect replays, and to
associate it with the message to which it replies. The KRB_ERROR
message also contains information which can be used to associate it
with the message to which it replies, but the lack of encryption in
the KRB_ERROR message precludes the ability to detect replays or
fabrications of such messages.

3.3.1. Generation of KRB_TGS_REQ message

Before sending a request to the ticket-granting service, the client
must determine in which realm the application server is registered
[Note: This can be accomplished in several ways. It might be known
beforehand (since the realm is part of the principal identifier), or
it might be stored in a nameserver. Presently, however, this
information is obtained from a configuration file. If the realm to
be used is obtained from a nameserver, there is a danger of being
spoofed if the nameservice providing the realm name is not
authenticated. This might result in the use of a realm which has
been compromised, and would result in an attacker's ability to
compromise the authentication of the application server to the
client.]. If the client does not already possess a ticket-granting
ticket for the appropriate realm, then one must be obtained. This is
first attempted by requesting a ticket-granting ticket for the
destination realm from the local Kerberos server (using the

KRB_TGS_REQ message recursively). The Kerberos server may return a
TGT for the desired realm in which case one can proceed.
Alternatively, the Kerberos server may return a TGT for a realm which
is "closer" to the desired realm (further along the standard
hierarchical path), in which case this step must be repeated with a
Kerberos server in the realm specified in the returned TGT. If
neither are returned, then the request must be retried with a
Kerberos server for a realm higher in the hierarchy. This request
will itself require a ticket-granting ticket for the higher realm
which must be obtained by recursively applying these directions.

Once the client obtains a ticket-granting ticket for the appropriate
realm, it determines which Kerberos servers serve that realm, and
contacts one. The list might be obtained through a configuration file
or network service; as long as the secret keys exchanged by realms
are kept secret, only denial of service results from a false Kerberos
server.

As in the AS exchange, the client may specify a number of options in
the KRB_TGS_REQ message. The client prepares the KRB_TGS_REQ
message, providing an authentication header as an element of the
padata field, and including the same fields as used in the KRB_AS_REQ
message along with several optional fields: the enc-authorization-
data field for application server use and additional tickets required
by some options.

In preparing the authentication header, the client can select a sub-
session key under which the response from the Kerberos server will be
encrypted (If the client selects a sub-session key, care must be
taken to ensure the randomness of the selected subsession key. One
approach would be to generate a random number and XOR it with the
session key from the ticket-granting ticket.). If the sub-session key
is not specified, the session key from the ticket-granting ticket
will be used. If the enc-authorization-data is present, it must be
encrypted in the sub-session key, if present, from the authenticator
portion of the authentication header, or if not present in the
session key from the ticket-granting ticket.

Once prepared, the message is sent to a Kerberos server for the
destination realm. See section A.5 for pseudocode.

3.3.2. Receipt of KRB_TGS_REQ message

The KRB_TGS_REQ message is processed in a manner similar to the
KRB_AS_REQ message, but there are many additional checks to be
performed. First, the Kerberos server must determine which server
the accompanying ticket is for and it must select the appropriate key
to decrypt it. For a normal KRB_TGS_REQ message, it will be for the

ticket granting service, and the TGS's key will be used. If the TGT
was issued by another realm, then the appropriate inter-realm key
must be used. If the accompanying ticket is not a ticket granting
ticket for the current realm, but is for an application server in the
current realm, the RENEW, VALIDATE, or PROXY options are specified in
the request, and the server for which a ticket is requested is the
server named in the accompanying ticket, then the KDC will decrypt
the ticket in the authentication header using the key of the server
for which it was issued. If no ticket can be found in the padata
field, the KDC_ERR_PADATA_TYPE_NOSUPP error is returned.

Once the accompanying ticket has been decrypted, the user-supplied
checksum in the Authenticator must be verified against the contents
of the request, and the message rejected if the checksums do not
match (with an error code of KRB_AP_ERR_MODIFIED) or if the checksum
is not keyed or not collision-proof (with an error code of
KRB_AP_ERR_INAPP_CKSUM). If the checksum type is not supported, the
KDC_ERR_SUMTYPE_NOSUPP error is returned. If the authorization-data
are present, they are decrypted using the sub-session key from the
Authenticator.

If any of the decryptions indicate failed integrity checks, the
KRB_AP_ERR_BAD_INTEGRITY error is returned.

3.3.3. Generation of KRB_TGS_REP message

The KRB_TGS_REP message shares its format with the KRB_AS_REP
(KRB_KDC_REP), but with its type field set to KRB_TGS_REP. The
detailed specification is in section 5.4.2.

The response will include a ticket for the requested server. The
Kerberos database is queried to retrieve the record for the requested
server (including the key with which the ticket will be encrypted).
If the request is for a ticket granting ticket for a remote realm,
and if no key is shared with the requested realm, then the Kerberos
server will select the realm "closest" to the requested realm with
which it does share a key, and use that realm instead. This is the
only case where the response from the KDC will be for a different
server than that requested by the client.

By default, the address field, the client's name and realm, the list
of transited realms, the time of initial authentication, the
expiration time, and the authorization data of the newly-issued
ticket will be copied from the ticket-granting ticket (TGT) or
renewable ticket. If the transited field needs to be updated, but
the transited type is not supported, the KDC_ERR_TRTYPE_NOSUPP error
is returned.

If the request specifies an endtime, then the endtime of the new
ticket is set to the minimum of (a) that request, (b) the endtime
from the TGT, and (c) the starttime of the TGT plus the minimum of
the maximum life for the application server and the maximum life for
the local realm (the maximum life for the requesting principal was
already applied when the TGT was issued). If the new ticket is to be
a renewal, then the endtime above is replaced by the minimum of (a)
the value of the renew_till field of the ticket and (b) the starttime
for the new ticket plus the life (endtimestarttime) of the old
ticket.

If the FORWARDED option has been requested, then the resulting ticket
will contain the addresses specified by the client. This option will
only be honored if the FORWARDABLE flag is set in the TGT. The PROXY
option is similar; the resulting ticket will contain the addresses
specified by the client. It will be honored only if the PROXIABLE
flag in the TGT is set. The PROXY option will not be honored on
requests for additional ticket-granting tickets.

If the requested start time is absent or indicates a time in the
past, then the start time of the ticket is set to the authentication
server's current time. If it indicates a time in the future, but the
POSTDATED option has not been specified or the MAY-POSTDATE flag is
not set in the TGT, then the error KDC_ERR_CANNOT_POSTDATE is
returned. Otherwise, if the ticket-granting ticket has the
MAYPOSTDATE flag set, then the resulting ticket will be postdated and
the requested starttime is checked against the policy of the local
realm. If acceptable, the ticket's start time is set as requested,
and the INVALID flag is set. The postdated ticket must be validated
before use by presenting it to the KDC after the starttime has been
reached. However, in no case may the starttime, endtime, or renew-
till time of a newly-issued postdated ticket extend beyond the
renew-till time of the ticket-granting ticket.

If the ENC-TKT-IN-SKEY option has been specified and an additional
ticket has been included in the request, the KDC will decrypt the
additional ticket using the key for the server to which the
additional ticket was issued and verify that it is a ticket-granting
ticket. If the name of the requested server is missing from the
request, the name of the client in the additional ticket will be
used. Otherwise the name of the requested server will be compared to
the name of the client in the additional ticket and if different, the
request will be rejected. If the request succeeds, the session key
from the additional ticket will be used to encrypt the new ticket
that is issued instead of using the key of the server for which the
new ticket will be used (This allows easy implementation of user-to-
user authentication [6], which uses ticket-granting ticket session
keys in lieu of secret server keys in situations where such secret

keys could be easily compromised.).

If the name of the server in the ticket that is presented to the KDC
as part of the authentication header is not that of the ticket-
granting server itself, and the server is registered in the realm of
the KDC, If the RENEW option is requested, then the KDC will verify
that the RENEWABLE flag is set in the ticket and that the renew_till
time is still in the future. If the VALIDATE option is rqeuested,
the KDC will check that the starttime has passed and the INVALID flag
is set. If the PROXY option is requested, then the KDC will check
that the PROXIABLE flag is set in the ticket. If the tests succeed,
the KDC will issue the appropriate new ticket.

Whenever a request is made to the ticket-granting server, the
presented ticket(s) is(are) checked against a hot-list of tickets
which have been canceled. This hot-list might be implemented by
storing a range of issue dates for "suspect tickets"; if a presented
ticket had an authtime in that range, it would be rejected. In this
way, a stolen ticket-granting ticket or renewable ticket cannot be
used to gain additional tickets (renewals or otherwise) once the
theft has been reported. Any normal ticket obtained before it was
reported stolen will still be valid (because they require no
interaction with the KDC), but only until their normal expiration
time.

The ciphertext part of the response in the KRB_TGS_REP message is
encrypted in the sub-session key from the Authenticator, if present,
or the session key key from the ticket-granting ticket. It is not
encrypted using the client's secret key. Furthermore, the client's
key's expiration date and the key version number fields are left out
since these values are stored along with the client's database
record, and that record is not needed to satisfy a request based on a
ticket-granting ticket. See section A.6 for pseudocode.

3.3.3.1. Encoding the transited field

If the identity of the server in the TGT that is presented to the KDC
as part of the authentication header is that of the ticket-granting
service, but the TGT was issued from another realm, the KDC will look
up the inter-realm key shared with that realm and use that key to
decrypt the ticket. If the ticket is valid, then the KDC will honor
the request, subject to the constraints outlined above in the section
describing the AS exchange. The realm part of the client's identity
will be taken from the ticket-granting ticket. The name of the realm
that issued the ticket-granting ticket will be added to the transited
field of the ticket to be issued. This is accomplished by reading
the transited field from the ticket-granting ticket (which is treated
as an unordered set of realm names), adding the new realm to the set,

then constructing and writing out its encoded (shorthand) form (this
may involve a rearrangement of the existing encoding).

Note that the ticket-granting service does not add the name of its
own realm. Instead, its responsibility is to add the name of the
previous realm. This prevents a malicious Kerberos server from
intentionally leaving out its own name (it could, however, omit other
realms' names).

The names of neither the local realm nor the principal's realm are to
be included in the transited field. They appear elsewhere in the
ticket and both are known to have taken part in authenticating the
principal. Since the endpoints are not included, both local and
single-hop inter-realm authentication result in a transited field
that is empty.

Because the name of each realm transited is added to this field,
it might potentially be very long. To decrease the length of this
field, its contents are encoded. The initially supported encoding is
optimized for the normal case of inter-realm communication: a
hierarchical arrangement of realms using either domain or X.500 style
realm names. This encoding (called DOMAIN-X500-COMPRESS) is now
described.

Realm names in the transited field are separated by a ",". The ",",
"\", trailing "."s, and leading spaces (" ") are special characters,
and if they are part of a realm name, they must be quoted in the
transited field by preceding them with a "\".

A realm name ending with a "." is interpreted as being prepended to
the previous realm. For example, we can encode traversal of EDU,
MIT.EDU, ATHENA.MIT.EDU, WASHINGTON.EDU, and CS.WASHINGTON.EDU as:

        "EDU,MIT.,ATHENA.,WASHINGTON.EDU,CS.".

Note that if ATHENA.MIT.EDU, or CS.WASHINGTON.EDU were endpoints,
that they would not be included in this field, and we would have:

        "EDU,MIT.,WASHINGTON.EDU"

A realm name beginning with a "/" is interpreted as being appended to
the previous realm (For the purpose of appending, the realm preceding
the first listed realm is considered to be the null realm ("")). If
it is to stand by itself, then it should be preceded by a space ("
"). For example, we can encode traversal of /COM/HP/APOLLO, /COM/HP,
/COM, and /COM/DEC as:

        "/COM,/HP,/APOLLO, /COM/DEC".

Like the example above, if /COM/HP/APOLLO and /COM/DEC are endpoints,
they they would not be included in this field, and we would have:

        "/COM,/HP"

A null subfield preceding or following a "," indicates that all
realms between the previous realm and the next realm have been
traversed (For the purpose of interpreting null subfields, the
client's realm is considered to precede those in the transited field,
and the server's realm is considered to follow them.). Thus, ","
means that all realms along the path between the client and the
server have been traversed. ",EDU, /COM," means that that all realms
from the client's realm up to EDU (in a domain style hierarchy) have
been traversed, and that everything from /COM down to the server's
realm in an X.500 style has also been traversed. This could occur if
the EDU realm in one hierarchy shares an inter-realm key directly
with the /COM realm in another hierarchy.

3.3.4. Receipt of KRB_TGS_REP message

When the KRB_TGS_REP is received by the client, it is processed in
the same manner as the KRB_AS_REP processing described above. The
primary difference is that the ciphertext part of the response must
be decrypted using the session key from the ticket-granting ticket
rather than the client's secret key. See section A.7 for pseudocode.

3.4.   The KRB_SAFE Exchange

The KRB_SAFE message may be used by clients requiring the ability to
detect modifications of messages they exchange. It achieves this by
including a keyed collisionproof checksum of the user data and some
control information. The checksum is keyed with an encryption key
(usually the last key negotiated via subkeys, or the session key if
no negotiation has occured).

3.4.1. Generation of a KRB_SAFE message

When an application wishes to send a KRB_SAFE message, it collects
its data and the appropriate control information and computes a
checksum over them. The checksum algorithm should be some sort of
keyed one-way hash function (such as the RSA-MD5-DES checksum
algorithm specified in section 6.4.5, or the DES MAC), generated
using the sub-session key if present, or the session key. Different
algorithms may be selected by changing the checksum type in the
message. Unkeyed or non-collision-proof checksums are not suitable
for this use.

The control information for the KRB_SAFE message includes both a

timestamp and a sequence number. The designer of an application
using the KRB_SAFE message must choose at least one of the two
mechanisms. This choice should be based on the needs of the
application protocol.

Sequence numbers are useful when all messages sent will be received
by one's peer. Connection state is presently required to maintain
the session key, so maintaining the next sequence number should not
present an additional problem.

If the application protocol is expected to tolerate lost messages
without them being resent, the use of the timestamp is the
appropriate replay detection mechanism. Using timestamps is also the
appropriate mechanism for multi-cast protocols where all of one's
peers share a common sub-session key, but some messages will be sent
to a subset of one's peers.

After computing the checksum, the client then transmits the
information and checksum to the recipient in the message format
specified in section 5.6.1.

3.4.2. Receipt of KRB_SAFE message

When an application receives a KRB_SAFE message, it verifies it as
follows. If any error occurs, an error code is reported for use by
the application.

The message is first checked by verifying that the protocol version
and type fields match the current version and KRB_SAFE, respectively.
A mismatch generates a KRB_AP_ERR_BADVERSION or KRB_AP_ERR_MSG_TYPE
error. The application verifies that the checksum used is a
collisionproof keyed checksum, and if it is not, a
KRB_AP_ERR_INAPP_CKSUM error is generated. The recipient verifies
that the operating system's report of the sender's address matches
the sender's address in the message, and (if a recipient address is
specified or the recipient requires an address) that one of the
recipient's addresses appears as the recipient's address in the
message. A failed match for either case generates a
KRB_AP_ERR_BADADDR error. Then the timestamp and usec and/or the
sequence number fields are checked. If timestamp and usec are
expected and not present, or they are present but not current, the
KRB_AP_ERR_SKEW error is generated. If the server name, along with
the client name, time and microsecond fields from the Authenticator
match any recently-seen such tuples, the KRB_AP_ERR_REPEAT error is
generated. If an incorrect sequence number is included, or a
sequence number is expected but not present, the KRB_AP_ERR_BADORDER
error is generated. If neither a timestamp and usec or a sequence
number is present, a KRB_AP_ERR_MODIFIED error is generated.

Finally, the checksum is computed over the data and control
information, and if it doesn't match the received checksum, a
KRB_AP_ERR_MODIFIED error is generated.

If all the checks succeed, the application is assured that the
message was generated by its peer and was not modified in transit.

3.5. The KRB_PRIV Exchange

The KRB_PRIV message may be used by clients requiring confidentiality
and the ability to detect modifications of exchanged messages. It
achieves this by encrypting the messages and adding control
information.

3.5.1. Generation of a KRB_PRIV message

When an application wishes to send a KRB_PRIV message, it collects
its data and the appropriate control information (specified in
section 5.7.1) and encrypts them under an encryption key (usually the
last key negotiated via subkeys, or the session key if no negotiation
has occured). As part of the control information, the client must
choose to use either a timestamp or a sequence number (or both); see
the discussion in section 3.4.1 for guidelines on which to use.
After the user data and control information are encrypted, the client
transmits the ciphertext and some "envelope" information to the
recipient.

3.5.2. Receipt of KRB_PRIV message

When an application receives a KRB_PRIV message, it verifies it as
follows. If any error occurs, an error code is reported for use by
the application.

The message is first checked by verifying that the protocol version
and type fields match the current version and KRB_PRIV, respectively.
A mismatch generates a KRB_AP_ERR_BADVERSION or KRB_AP_ERR_MSG_TYPE
error. The application then decrypts the ciphertext and processes
the resultant plaintext. If decryption shows the data to have been
modified, a KRB_AP_ERR_BAD_INTEGRITY error is generated. The
recipient verifies that the operating system's report of the sender's
address matches the sender's address in the message, and (if a
recipient address is specified or the recipient requires an address)
that one of the recipient's addresses appears as the recipient's
address in the message. A failed match for either case generates a
KRB_AP_ERR_BADADDR error. Then the timestamp and usec and/or the
sequence number fields are checked. If timestamp and usec are
expected and not present, or they are present but not current, the
KRB_AP_ERR_SKEW error is generated. If the server name, along with

the client name, time and microsecond fields from the Authenticator
match any recently-seen such tuples, the KRB_AP_ERR_REPEAT error is
generated. If an incorrect sequence number is included, or a
sequence number is expected but not present, the KRB_AP_ERR_BADORDER
error is generated. If neither a timestamp and usec or a sequence
number is present, a KRB_AP_ERR_MODIFIED error is generated.

If all the checks succeed, the application can assume the message was
generated by its peer, and was securely transmitted (without
intruders able to see the unencrypted contents).

3.6. The KRB_CRED Exchange

The KRB_CRED message may be used by clients requiring the ability to
send Kerberos credentials from one host to another. It achieves this
by sending the tickets together with encrypted data containing the
session keys and other information associated with the tickets.

3.6.1. Generation of a KRB_CRED message

When an application wishes to send a KRB_CRED message it first (using
the KRB_TGS exchange) obtains credentials to be sent to the remote
host. It then constructs a KRB_CRED message using the ticket or
tickets so obtained, placing the session key needed to use each
ticket in the key field of the corresponding KrbCredInfo sequence of
the encrypted part of the the KRB_CRED message.

Other information associated with each ticket and obtained during the
KRB_TGS exchange is also placed in the corresponding KrbCredInfo
sequence in the encrypted part of the KRB_CRED message. The current
time and, if specifically required by the application the nonce, s-
address, and raddress fields, are placed in the encrypted part of the
KRB_CRED message which is then encrypted under an encryption key
previosuly exchanged in the KRB_AP exchange (usually the last key
negotiated via subkeys, or the session key if no negotiation has
occured).

3.6.2. Receipt of KRB_CRED message

When an application receives a KRB_CRED message, it verifies it. If
any error occurs, an error code is reported for use by the
application. The message is verified by checking that the protocol
version and type fields match the current version and KRB_CRED,
respectively. A mismatch generates a KRB_AP_ERR_BADVERSION or
KRB_AP_ERR_MSG_TYPE error. The application then decrypts the
ciphertext and processes the resultant plaintext. If decryption shows
the data to have been modified, a KRB_AP_ERR_BAD_INTEGRITY error is
generated.

If present or required, the recipient verifies that the operating
system's report of the sender's address matches the sender's address
in the message, and that one of the recipient's addresses appears as
the recipient's address in the message. A failed match for either
case generates a KRB_AP_ERR_BADADDR error. The timestamp and usec
fields (and the nonce field if required) are checked next. If the
timestamp and usec are not present, or they are present but not
current, the KRB_AP_ERR_SKEW error is generated.

If all the checks succeed, the application stores each of the new
tickets in its ticket cache together with the session key and other
information in the corresponding KrbCredInfo sequence from the
encrypted part of the KRB_CRED message.

4. The Kerberos Database

   The Kerberos server must have access to a database containing the
   principal identifiers and secret keys of principals to be
   authenticated (The implementation of the Kerberos server need not
   combine the database and the server on the same machine; it is
   feasible to store the principal database in, say, a network name
   service, as long as the entries stored therein are protected from
   disclosure to and modification by unauthorized parties. However, we
   recommend against such strategies, as they can make system management
   and threat analysis quite complex.).

4.1. Database contents

   A database entry should contain at least the following fields:

   Field   Value

   name   Principal's identifier
   key   Principal's secret key
   p_kvno   Principal's key version
   max_life   Maximum lifetime for Tickets
   max_renewable_life Maximum total lifetime for renewable
                       Tickets

   The name field is an encoding of the principal's identifier. The key
   field contains an encryption key. This key is the principal's secret
   key. (The key can be encrypted before storage under a Kerberos
   "master key" to protect it in case the database is compromised but
   the master key is not. In that case, an extra field must be added to
   indicate the master key version used, see below.) The p_kvno field is
   the key version number of the principal's secret key. The max_life
   field contains the maximum allowable lifetime (endtime - starttime)
   for any Ticket issued for this principal. The max_renewable_life

field contains the maximum allowable total lifetime for any renewable
Ticket issued for this principal. (See section 3.1 for a description
of how these lifetimes are used in determining the lifetime of a
given Ticket.)

A server may provide KDC service to several realms, as long as the
database representation provides a mechanism to distinguish between
principal records with identifiers which differ only in the realm
name.

When an application server's key changes, if the change is routine
(i.e., not the result of disclosure of the old key), the old key
should be retained by the server until all tickets that had been
issued using that key have expired. Because of this, it is possible
for several keys to be active for a single principal. Ciphertext
encrypted in a principal's key is always tagged with the version of
the key that was used for encryption, to help the recipient find the
proper key for decryption.

When more than one key is active for a particular principal, the
principal will have more than one record in the Kerberos database.
The keys and key version numbers will differ between the records (the
rest of the fields may or may not be the same). Whenever Kerberos
issues a ticket, or responds to a request for initial authentication,
the most recent key (known by the Kerberos server) will be used for
encryption. This is the key with the highest key version number.

4.2. Additional fields

Project Athena's KDC implementation uses additional fields in its
database:

Field   Value

K_kvno   Kerberos' key version
expiration   Expiration date for entry
attributes   Bit field of attributes
mod_date   Timestamp of last modification
mod_name   Modifying principal's identifier

The K_kvno field indicates the key version of the Kerberos master key
under which the principal's secret key is encrypted.

After an entry's expiration date has passed, the KDC will return an
error to any client attempting to gain tickets as or for the
principal. (A database may want to maintain two expiration dates:
one for the principal, and one for the principal's current key. This
allows password aging to work independently of the principal's

expiration date. However, due to the limited space in the responses,
the KDC must combine the key expiration and principal expiration date
into a single value called "key_exp", which is used as a hint to the
user to take administrative action.)

The attributes field is a bitfield used to govern the operations
involving the principal. This field might be useful in conjunction
with user registration procedures, for site-specific policy
implementations (Project Athena currently uses it for their user
registration process controlled by the system-wide database service,
Moira [7]), or to identify the "string to key" conversion algorithm
used for a principal's key. (See the discussion of the padata field
in section 5.4.2 for details on why this can be useful.) Other bits
are used to indicate that certain ticket options should not be
allowed in tickets encrypted under a principal's key (one bit each):
Disallow issuing postdated tickets, disallow issuing forwardable
tickets, disallow issuing tickets based on TGT authentication,
disallow issuing renewable tickets, disallow issuing proxiable
tickets, and disallow issuing tickets for which the principal is the
server.

The mod_date field contains the time of last modification of the
entry, and the mod_name field contains the name of the principal
which last modified the entry.

4.3. Frequently Changing Fields

Some KDC implementations may wish to maintain the last time that a
request was made by a particular principal. Information that might
be maintained includes the time of the last request, the time of the
last request for a ticket-granting ticket, the time of the last use
of a ticket-granting ticket, or other times. This information can
then be returned to the user in the last-req field (see section 5.2).

Other frequently changing information that can be maintained is the
latest expiration time for any tickets that have been issued using
each key. This field would be used to indicate how long old keys
must remain valid to allow the continued use of outstanding tickets.

4.4. Site Constants

The KDC implementation should have the following configurable
constants or options, to allow an administrator to make and enforce
policy decisions:

+ The minimum supported lifetime (used to determine whether the
  KDC_ERR_NEVER_VALID error should be returned). This constant
  should reflect reasonable expectations of round-trip time to the

KDC, encryption/decryption time, and processing time by the client
and target server, and it should allow for a minimum "useful"
lifetime.

+ The maximum allowable total (renewable) lifetime of a ticket
  (renew_till - starttime).

+ The maximum allowable lifetime of a ticket (endtime - starttime).

+ Whether to allow the issue of tickets with empty address fields
  (including the ability to specify that such tickets may only be
  issued if the request specifies some authorization_data).

+ Whether proxiable, forwardable, renewable or post-datable tickets
  are to be issued.

5. Message Specifications

   The following sections describe the exact contents and encoding of
   protocol messages and objects. The ASN.1 base definitions are
   presented in the first subsection. The remaining subsections specify
   the protocol objects (tickets and authenticators) and messages.
   Specification of encryption and checksum techniques, and the fields
   related to them, appear in section 6.

5.1. ASN.1 Distinguished Encoding Representation

   All uses of ASN.1 in Kerberos shall use the Distinguished Encoding
   Representation of the data elements as described in the X.509
   specification, section 8.7 [8].

5.2. ASN.1 Base Definitions

   The following ASN.1 base definitions are used in the rest of this
   section. Note that since the underscore character (_) is not
   permitted in ASN.1 names, the hyphen (-) is used in its place for the
   purposes of ASN.1 names.

   Realm ::=   GeneralString
   PrincipalName ::=   SEQUENCE {
                        name-type[0]   INTEGER,
                        name-string[1]   SEQUENCE OF GeneralString
   }

   Kerberos realms are encoded as GeneralStrings. Realms shall not
   contain a character with the code 0 (the ASCII NUL). Most realms
   will usually consist of several components separated by periods (.),
   in the style of Internet Domain Names, or separated by slashes (/) in

the style of X.500 names. Acceptable forms for realm names are
specified in section 7. A PrincipalName is a typed sequence of
components consisting of the following sub-fields:

name-type This field specifies the type of name that follows.
          Pre-defined values for this field are
          specified in section 7.2. The name-type should be
          treated as a hint. Ignoring the name type, no two
          names can be the same (i.e., at least one of the
          components, or the realm, must be different).
          This constraint may be eliminated in the future.

name-string This field encodes a sequence of components that
          form a name, each component encoded as a General
          String. Taken together, a PrincipalName and a Realm
          form a principal identifier. Most PrincipalNames
          will have only a few components (typically one or two).

          KerberosTime ::=   GeneralizedTime
                             -- Specifying UTC time zone (Z)

The timestamps used in Kerberos are encoded as GeneralizedTimes. An
encoding shall specify the UTC time zone (Z) and shall not include
any fractional portions of the seconds. It further shall not include
any separators. Example: The only valid format for UTC time 6
minutes, 27 seconds after 9 pm on 6 November 1985 is 19851106210627Z.

```
 HostAddress ::=       SEQUENCE  {
                       addr-type[0]            INTEGER,
                       address[1]              OCTET STRING
 }

 HostAddresses ::=     SEQUENCE OF SEQUENCE {
                       addr-type[0]            INTEGER,
                       address[1]              OCTET STRING
 }
```

The host adddress encodings consists of two fields:

addr-type This field specifies the type of address that
           follows. Pre-defined values for this field are
           specified in section 8.1.

address   This field encodes a single address of type addr-type.

The two forms differ slightly. HostAddress contains exactly one

address; HostAddresses contains a sequence of possibly many
addresses.

```
AuthorizationData ::=    SEQUENCE OF SEQUENCE {
                     ad-type[0]                  INTEGER,
                     ad-data[1]                  OCTET STRING
}
```

ad-data   This field contains authorization data to be
            interpreted according to the value of the
            corresponding ad-type field.

ad-type   This field specifies the format for the ad-data
            subfield. All negative values are reserved for
            local use. Non-negative values are reserved for
            registered use.

```
                APOptions ::=   BIT STRING {
                            reserved(0),
                            use-session-key(1),
                            mutual-required(2)
                }


                TicketFlags ::=   BIT STRING {
                            reserved(0),
                            forwardable(1),
                            forwarded(2),
                            proxiable(3),
                            proxy(4),
                            may-postdate(5),
                            postdated(6),
                            invalid(7),
                            renewable(8),
                            initial(9),
                            pre-authent(10),
                            hw-authent(11)
                }

                KDCOptions ::=   BIT STRING {
                            reserved(0),
                            forwardable(1),
                            forwarded(2),
                            proxiable(3),
                            proxy(4),
                            allow-postdate(5),
                            postdated(6),
```

```
                                    unused7(7),
                                    renewable(8),
                                    unused9(9),
                                    unused10(10),
                                    unused11(11),
                                    renewable-ok(27),
                                    enc-tkt-in-skey(28),
                                    renew(30),
                                    validate(31)
              }


        LastReq ::=    SEQUENCE OF SEQUENCE {
                    lr-type[0]                  INTEGER,
                    lr-value[1]                 KerberosTime
              }
```

   lr-type   This field indicates how the following lr-value
             field is to be interpreted. Negative values indicate
             that the information pertains only to the
             responding server. Non-negative values pertain to
             all servers for the realm.

             If the lr-type field is zero (0), then no information
             is conveyed by the lr-value subfield. If the
             absolute value of the lr-type field is one (1),
             then the lr-value subfield is the time of last
             initial request for a TGT. If it is two (2), then
             the lr-value subfield is the time of last initial
             request. If it is three (3), then the lr-value
             subfield is the time of issue for the newest
             ticket-granting ticket used. If it is four (4),
             then the lr-value subfield is the time of the last
             renewal. If it is five (5), then the lr-value
             subfield is the time of last request (of any
             type).

   lr-value This field contains the time of the last request.
             The time must be interpreted according to the contents
             of the accompanying lr-type subfield.

   See section 6 for the definitions of Checksum, ChecksumType,
   EncryptedData, EncryptionKey, EncryptionType, and KeyType.

   5.3. Tickets and Authenticators

      This section describes the format and encryption parameters for
      tickets and authenticators. When a ticket or authenticator is
      included in a protocol message it is treated as an opaque object.

   5.3.1. Tickets

      A ticket is a record that helps a client authenticate to a service.
      A Ticket contains the following information:

```
Ticket ::=                      [APPLICATION 1] SEQUENCE {
                                tkt-vno[0]                   INTEGER,
                                realm[1]                     Realm,
                                sname[2]                     PrincipalName,
                                enc-part[3]                  EncryptedData
}
-- Encrypted part of ticket
EncTicketPart ::=       [APPLICATION 3] SEQUENCE {
                        flags[0]            TicketFlags,
                        key[1]              EncryptionKey,
                        crealm[2]           Realm,
                        cname[3]            PrincipalName,
                        transited[4]        TransitedEncoding,
                        authtime[5]         KerberosTime,
                        starttime[6]        KerberosTime OPTIONAL,
                        endtime[7]          KerberosTime,
                        renew-till[8]       KerberosTime OPTIONAL,
                        caddr[9]            HostAddresses OPTIONAL,
                        authorization-data[10]   AuthorizationData OPTIONAL
}
-- encoded Transited field
TransitedEncoding ::=           SEQUENCE {
                        tr-type[0]  INTEGER, -- must be registered
                        contents[1]        OCTET STRING
}
```

      The encoding of EncTicketPart is encrypted in the key shared by
      Kerberos and the end server (the server's secret key). See section 6
      for the format of the ciphertext.

      tkt-vno   This field specifies the version number for the ticket
                format. This document describes version number 5.

      realm   This field specifies the realm that issued a ticket. It
                also serves to identify the realm part of the server's
                principal identifier. Since a Kerberos server can only
                issue tickets for servers within its realm, the two will

always be identical.

sname   This field specifies the name part of the server's
           identity.

enc-part   This field holds the encrypted encoding of the
           EncTicketPart sequence.

flags   This field indicates which of various options were used or
           requested when the ticket was issued. It is a bit-field,
           where the selected options are indicated by the bit being
           set (1), and the unselected options and reserved fields
           being reset (0). Bit 0 is the most significant bit. The
           encoding of the bits is specified in section 5.2. The
           flags are described in more detail above in section 2. The
           meanings of the flags are:

           Bit(s)   Name   Description

           0   RESERVED   Reserved for future expansion of this
                                   field.

           1   FORWARDABLE   The FORWARDABLE flag is normally only
                                   interpreted by the TGS, and can be
                                   ignored by end servers. When set,
                                   this flag tells the ticket-granting
                                   server that it is OK to issue a new
                                   ticket- granting ticket with a
                                   different network address based on
                                   the presented ticket.

           2   FORWARDED   When set, this flag indicates that
                                   the ticket has either been forwarded
                                   or was issued based on authentication
                                   involving a forwarded ticket-granting
                                   ticket.

           3   PROXIABLE   The PROXIABLE flag is normally only
                                   interpreted by the TGS, and can be
                                   ignored by end servers. The PROXIABLE
                                   flag has an interpretation identical
                                   to that of the FORWARDABLE flag,
                                   except that the PROXIABLE flag tells
                                   the ticket-granting server that only
                                   non- ticket-granting tickets may be
                                   issued with different network
                                   addresses.

4   PROXY   When set, this flag indicates that a
                         ticket is a proxy.

5   MAY-POSTDATE The MAY-POSTDATE flag is normally
                         only interpreted by the TGS, and can
                         be ignored by end servers. This flag
                         tells the ticket-granting server that
                         a post- dated ticket may be issued
                         based on this ticket-granting ticket.

6   POSTDATED   This flag indicates that this ticket
                         has been postdated. The end-service
                         can check the authtime field to see
                         when the original authentication
                         occurred.

7   INVALID   This flag indicates that a ticket is
                         invalid, and it must be validated by
                         the KDC before use. Application
                         servers must reject tickets which
                         have this flag set.

8   RENEWABLE   The RENEWABLE flag is normally only
                         interpreted by the TGS, and can
                         usually be ignored by end servers
                         (some particularly careful servers
                         may wish to disallow renewable
                         tickets). A renewable ticket can be
                         used to obtain a replacement ticket
                         that expires at a later date.

9   INITIAL   This flag indicates that this ticket
                         was issued using the AS protocol, and
                         not issued based on a ticket-granting
                         ticket.

10   PRE-AUTHENT This flag indicates that during
                         initial authentication, the client
                         was authenticated by the KDC before a
                         ticket was issued. The strength of
                         the preauthentication method is not
                         indicated, but is acceptable to the
                         KDC.

11   HW-AUTHENT   This flag indicates that the protocol
                         employed for initial authentication
                         required the use of hardware expected
                         to be possessed solely by the named

                               client. The hardware authentication
                               method is selected by the KDC and the
                               strength of the method is not
                               indicated.

              12-31   RESERVED   Reserved for future use.

    key   This field exists in the ticket and the KDC response and is
               used to pass the session key from Kerberos to the
               application server and the client. The field's encoding is
               described in section 6.2.

    crealm   This field contains the name of the realm in which the
               client is registered and in which initial authentication
               took place.

    cname   This field contains the name part of the client's principal
               identifier.

    transited This field lists the names of the Kerberos realms that took
               part in authenticating the user to whom this ticket was
               issued. It does not specify the order in which the realms
               were transited. See section 3.3.3.1 for details on how
               this field encodes the traversed realms.

    authtime   This field indicates the time of initial authentication for
               the named principal. It is the time of issue for the
               original ticket on which this ticket is based. It is
               included in the ticket to provide additional information to
               the end service, and to provide the necessary information
               for implementation of a 'hot list' service at the KDC.   An
               end service that is particularly paranoid could refuse to
               accept tickets for which the initial authentication
               occurred "too far" in the past.

               This field is also returned as part of the response from
               the KDC. When returned as part of the response to initial
               authentication (KRB_AS_REP), this is the current time on
               the Kerberos server (It is NOT recommended that this time
               value be used to adjust the workstation's clock since the
               workstation cannot reliably determine that such a
               KRB_AS_REP actually came from the proper KDC in a timely
               manner.).

    starttime This field in the ticket specifies the time after which the
               ticket is valid. Together with endtime, this field
               specifies the life of the ticket.   If it is absent from
               the ticket, its value should be treated as that of the

                authtime field.

     endtime   This field contains the time after which the ticket will
                not be honored (its expiration time). Note that individual
                services may place their own limits on the life of a ticket
                and may reject tickets which have not yet expired. As
                such, this is really an upper bound on the expiration time
                for the ticket.

     renew-till This field is only present in tickets that have the
                RENEWABLE flag set in the flags field. It indicates the
                maximum endtime that may be included in a renewal. It can
                be thought of as the absolute expiration time for the
                ticket, including all renewals.

     caddr   This field in a ticket contains zero (if omitted) or more
                (if present) host addresses. These are the addresses from
                which the ticket can be used. If there are no addresses,
                the ticket can be used from any location. The decision
                by the KDC to issue or by the end server to accept zero-
                address tickets is a policy decision and is left to the
                Kerberos and end-service administrators; they may refuse to
                issue or accept such tickets. The suggested and default
                policy, however, is that such tickets will only be issued
                or accepted when additional information that can be used to
                restrict the use of the ticket is included in the
                authorization_data field. Such a ticket is a capability.

                Network addresses are included in the ticket to make it
                harder for an attacker to use stolen credentials. Because
                the session key is not sent over the network in cleartext,
                credentials can't be stolen simply by listening to the
                network; an attacker has to gain access to the session key
                (perhaps through operating system security breaches or a
                careless user's unattended session) to make use of stolen
                tickets.

                It is important to note that the network address from which
                a connection is received cannot be reliably determined.
                Even if it could be, an attacker who has compromised the
                client's workstation could use the credentials from there.
                Including the network addresses only makes it more
                difficult, not impossible, for an attacker to walk off with
                stolen credentials and then use them from a "safe"
                location.

authorization-data The authorization-data field is used to pass
          authorization data from the principal on whose behalf a
          ticket was issued to the application service. If no
          authorization data is included, this field will be left
          out. The data in this field are specific to the end
          service. It is expected that the field will contain the
          names of service specific objects, and the rights to those
          objects. The format for this field is described in section
          5.2. Although Kerberos is not concerned with the format of
          the contents of the subfields, it does carry type
          information (ad-type).

          By using the authorization_data field, a principal is able
          to issue a proxy that is valid for a specific purpose. For
          example, a client wishing to print a file can obtain a file
          server proxy to be passed to the print server. By
          specifying the name of the file in the authorization_data
          field, the file server knows that the print server can only
          use the client's rights when accessing the particular file
          to be printed.

          It is interesting to note that if one specifies the
          authorization-data field of a proxy and leaves the host
          addresses blank, the resulting ticket and session key can
          be treated as a capability. See [9] for some suggested
          uses of this field.

          The authorization-data field is optional and does not have
          to be included in a ticket.

5.3.2. Authenticators

   An authenticator is a record sent with a ticket to a server to
   certify the client's knowledge of the encryption key in the ticket,
   to help the server detect replays, and to help choose a "true session
   key" to use with the particular session. The encoding is encrypted
   in the ticket's session key shared by the client and the server:

```
-- Unencrypted authenticator
Authenticator ::=    [APPLICATION 2] SEQUENCE    {
             authenticator-vno[0]            INTEGER,
             crealm[1]                       Realm,
             cname[2]                        PrincipalName,
             cksum[3]                        Checksum OPTIONAL,
             cusec[4]                        INTEGER,
             ctime[5]                        KerberosTime,
             subkey[6]                       EncryptionKey OPTIONAL,
             seq-number[7]                   INTEGER OPTIONAL,
```

                    authorization-data[8]   AuthorizationData OPTIONAL
                       }

   authenticator-vno This field specifies the version number for the
             format of the authenticator. This document specifies
             version 5.

   crealm and cname These fields are the same as those described for the
             ticket in section 5.3.1.

   cksum   This field contains a checksum of the the application data
             that accompanies the KRB_AP_REQ.

   cusec   This field contains the microsecond part of the client's
             timestamp. Its value (before encryption) ranges from 0 to
             999999. It often appears along with ctime. The two fields
             are used together to specify a reasonably accurate
             timestamp.

   ctime   This field contains the current time on the client's host.

   subkey   This field contains the client's choice for an encryption
             key which is to be used to protect this specific
             application session. Unless an application specifies
             otherwise, if this field is left out the session key from
             the ticket will be used.

   seq-number This optional field includes the initial sequence number
             to be used by the KRB_PRIV or KRB_SAFE messages when
             sequence numbers are used to detect replays (It may also be
             used by application specific messages). When included in
             the authenticator this field specifies the initial sequence
             number for messages from the client to the server. When
             included in the AP-REP message, the initial sequence number
             is that for messages from the server to the client. When
             used in KRB_PRIV or KRB_SAFE messages, it is incremented by
             one after each message is sent.

             For sequence numbers to adequately support the detection of
             replays they should be non-repeating, even across
             connection boundaries. The initial sequence number should
             be random and uniformly distributed across the full space
             of possible sequence numbers, so that it cannot be guessed
             by an attacker and so that it and the successive sequence
             numbers do not repeat other sequences.

          authorization-data This field is the same as described for the ticket
                    in section 5.3.1. It is optional and will only appear when
                    additional restrictions are to be placed on the use of a
                    ticket, beyond those carried in the ticket itself.

5.4. Specifications for the AS and TGS exchanges

     This section specifies the format of the messages used in exchange
     between the client and the Kerberos server. The format of possible
     error messages appears in section 5.9.1.

5.4.1. KRB_KDC_REQ definition

     The KRB_KDC_REQ message has no type of its own. Instead, its type is
     one of KRB_AS_REQ or KRB_TGS_REQ depending on whether the request is
     for an initial ticket or an additional ticket. In either case, the
     message is sent from the client to the Authentication Server to
     request credentials for a service.

The message fields are:

AS-REQ ::=   [APPLICATION 10] KDC-REQ
TGS-REQ ::=   [APPLICATION 12] KDC-REQ

KDC-REQ ::=   SEQUENCE {
          pvno[1]   INTEGER,
          msg-type[2]   INTEGER,
          padata[3]   SEQUENCE OF PA-DATA OPTIONAL,
          req-body[4]   KDC-REQ-BODY
}

PA-DATA ::=   SEQUENCE {
          padata-type[1]   INTEGER,
          padata-value[2]   OCTET STRING,
                         -- might be encoded AP-REQ
}

KDC-REQ-BODY ::=   SEQUENCE {
          kdc-options[0]   KDCOptions,
          cname[1]   PrincipalName OPTIONAL,
                         -- Used only in AS-REQ
          realm[2]   Realm, -- Server's realm
                         -- Also client's in AS-REQ
          sname[3]   PrincipalName OPTIONAL,
          from[4]   KerberosTime OPTIONAL,
          till[5]   KerberosTime,
          rtime[6]   KerberosTime OPTIONAL,
          nonce[7]   INTEGER,

```
            etype[8]                SEQUENCE OF INTEGER, -- EncryptionType,
                        -- in preference order
            addresses[9]            HostAddresses OPTIONAL,
            enc-authorization-data[10]    EncryptedData OPTIONAL,
                        -- Encrypted AuthorizationData encoding
            additional-tickets[11]        SEQUENCE OF Ticket OPTIONAL
}
```

The fields in this message are:

pvno   This field is included in each message, and specifies the
            protocol version number. This document specifies protocol
            version 5.

msg-type   This field indicates the type of a protocol message. It
            will almost always be the same as the application
            identifier associated with a message. It is included to
            make the identifier more readily accessible to the
            application. For the KDC-REQ message, this type will be
            KRB_AS_REQ or KRB_TGS_REQ.

padata   The padata (pre-authentication data) field contains a of
            authentication information which may be needed before
            credentials can be issued or decrypted. In the case of
            requests for additional tickets (KRB_TGS_REQ), this field
            will include an element with padata-type of PA-TGS-REQ and
            data of an authentication header (ticket-granting ticket
            and authenticator). The checksum in the authenticator
            (which must be collisionproof) is to be computed over the
            KDC-REQ-BODY encoding. In most requests for initial
            authentication (KRB_AS_REQ) and most replies (KDC-REP), the
            padata field will be left out.

            This field may also contain information needed by certain
            extensions to the Kerberos protocol. For example, it might
            be used to initially verify the identity of a client before
            any response is returned. This is accomplished with a
            padata field with padata-type equal to PA-ENC-TIMESTAMP and
            padata-value defined as follows:

    padata-type   ::= PA-ENC-TIMESTAMP
    padata-value   ::= EncryptedData -- PA-ENC-TS-ENC

    PA-ENC-TS-ENC   ::= SEQUENCE {
            patimestamp[0]   KerberosTime, -- client's time
            pausec[1]   INTEGER OPTIONAL
    }
```

with patimestamp containing the client's time and pausec
containing the microseconds which may be omitted if a
client will not generate more than one request per second.
The ciphertext (padata-value) consists of the PA-ENC-TS-ENC
sequence, encrypted using the client's secret key.

The padata field can also contain information needed to
help the KDC or the client select the key needed for
generating or decrypting the response.  This form of the
padata is useful for supporting the use of certain
"smartcards" with Kerberos.  The details of such extensions
are beyond the scope of this specification.  See [10] for
additional uses of this field.

padata-type The padata-type element of the padata field indicates the
            way that the padata-value element is to be interpreted.
            Negative values of padata-type are reserved for
            unregistered use; non-negative values are used for a
            registered interpretation of the element type.

req-body   This field is a placeholder delimiting the extent of the
           remaining fields.  If a checksum is to be calculated over
           the request, it is calculated over an encoding of the KDC-
           REQ-BODY sequence which is enclosed within the req-body
           field.

kdc-options This field appears in the KRB_AS_REQ and KRB_TGS_REQ
            requests to the KDC and indicates the flags that the client
            wants set on the tickets as well as other information that
            is to modify the behavior of the KDC. Where appropriate,
            the name of an option may be the same as the flag that is
            set by that option.  Although in most case, the bit in the
            options field will be the same as that in the flags field,
            this is not guaranteed, so it is not acceptable to simply
            copy the options field to the flags field.  There are
            various checks that must be made before honoring an option
            anyway.

            The kdc_options field is a bit-field, where the selected
            options are indicated by the bit being set (1), and the
            unselected options and reserved fields being reset (0).
            The encoding of the bits is specified in section 5.2.  The
            options are described in more detail above in section 2.
            The meanings of the options are:

|        Bit(s)   Name           Description |

| Bit(s) | Name | Description |
|--------|------|-------------|
| 0 | RESERVED | Reserved for future expansion of this field. |
| 1 | FORWARDABLE | The FORWARDABLE option indicates that the ticket to be issued is to have its forwardable flag set.  It may only be set on the initial request, or in a subsequent request if the ticket-granting ticket on which it is based is also forwardable. |
| 2 | FORWARDED | The FORWARDED option is only specified in a request to the ticket-granting server and will only be honored if the ticket-granting ticket in the request has its FORWARDABLE bit set.  This option indicates that this is a request for forwarding. The address(es) of the host from which the resulting ticket is to be valid are included in the addresses field of the request. |
| 3 | PROXIABLE | The PROXIABLE option indicates that the ticket to be issued is to have its proxiable flag set. It may only be set on the initial request, or in a subsequent request if the ticket-granting ticket on which it is based is also proxiable. |
| 4 | PROXY | The PROXY option indicates that this is a request for a proxy.  This option will only be honored if the ticket-granting ticket in the request has its PROXIABLE bit set.  The address(es) of the host from which the resulting ticket is to be valid are included in the addresses field of the request. |
| 5 | ALLOW-POSTDATE | The ALLOW-POSTDATE option indicates that the ticket to be issued is to have its MAY-POSTDATE flag set.  It may only be set on the initial request, or in a subsequent request if |

the ticket-granting ticket on which it
is based also has its MAY-POSTDATE
flag set.

6        POSTDATED      The POSTDATED option indicates that
this is a request for a postdated
ticket.  This option will only be
honored if the ticket-granting ticket
on which it is based has its MAY-
POSTDATE flag set.  The resulting
ticket will also have its INVALID flag
set, and that flag may be reset by a
subsequent request to the KDC after
the starttime in the ticket has been
reached.

7        UNUSED         This option is presently unused.

8        RENEWABLE      The RENEWABLE option indicates that
the ticket to be issued is to have its
RENEWABLE flag set.  It may only be
set on the initial request, or when
the ticket-granting ticket on which
the request is based is also
renewable.  If this option is
requested, then the rtime field in the
request contains the desired absolute
expiration time for the ticket.

9-26     RESERVED       Reserved for future use.

27       RENEWABLE-OK   The RENEWABLE-OK option indicates that
a renewable ticket will be acceptable
if a ticket with the requested life
cannot otherwise be provided.  If a
ticket with the requested life cannot
be provided, then a renewable ticket
may be issued with a renew-till equal
to the the requested endtime.  The
value of the renew-till field may
still be limited by local limits, or
limits selected by the individual
principal or server.

28       ENC-TKT-IN-SKEY This option is used only by the
ticket-granting service.  The ENC-
TKT-IN-SKEY option indicates that the
ticket for the end server is to be

                              encrypted in the session key from the
                              additional ticket-granting ticket
                              provided.

               29      RESERVED      Reserved for future use.

               30      RENEW         This option is used only by the
                                     ticket-granting service.  The RENEW
                                     option indicates that the present
                                     request is for a renewal.  The ticket
                                     provided is encrypted in the secret
                                     key for the server on which it is
                                     valid.  This option will only be
                                     honored if the ticket to be renewed
                                     has its RENEWABLE flag set and if the
                                     time in its renew till field has not
                                     passed.  The ticket to be renewed is
                                     passed in the padata field as part of
                                     the authentication header.

               31      VALIDATE      This option is used only by the
                                     ticket-granting service.  The VALIDATE
                                     option indicates that the request is
                                     to validate a postdated ticket.  It
                                     will only be honored if the ticket
                                     presented is postdated, presently has
                                     its INVALID flag set, and would be
                                     otherwise usable at this time.  A
                                     ticket cannot be validated before its
                                     starttime.  The ticket presented for
                                     validation is encrypted in the key of
                                     the server for which it is valid and
                                     is passed in the padata field as part
                                     of the authentication header.

      cname and sname These fields are the same as those described for the
               ticket in section 5.3.1.  sname may only be absent when the
               ENC-TKT-IN-SKEY option is specified.  If absent, the name
               of the server is taken from the name of the client in the
               ticket passed as additional-tickets.

      enc-authorization-data The enc-authorization-data, if present (and it
               can only be present in the TGS_REQ form), is an encoding of
               the desired authorization-data encrypted under the sub-
               session key if present in the Authenticator, or
               alternatively from the session key in the ticket-granting
               ticket, both from the padata field in the KRB_AP_REQ.

realm      This field specifies the realm part of the server's
           principal identifier. In the AS exchange, this is also the
           realm part of the client's principal identifier.

from       This field is included in the KRB_AS_REQ and KRB_TGS_REQ
           ticket requests when the requested ticket is to be
           postdated.  It specifies the desired start time for the
           requested ticket.

till       This field contains the expiration date requested by the
           client in a ticket request.

rtime      This field is the requested renew-till time sent from a
           client to the KDC in a ticket request.  It is optional.

nonce      This field is part of the KDC request and response.  It it
           intended to hold a random number generated by the client.
           If the same number is included in the encrypted response
           from the KDC, it provides evidence that the response is
           fresh and has not been replayed by an attacker.  Nonces
           must never be re-used.  Ideally, it should be gen erated
           randomly, but if the correct time is known, it may suffice
           (Note, however, that if the time is used as the nonce, one
           must make sure that the workstation time is monotonically
           increasing.  If the time is ever reset backwards, there is
           a small, but finite, probability that a nonce will be
           reused.).

etype      This field specifies the desired encryption algorithm to be
           used in the response.

addresses  This field is included in the initial request for tickets,
           and optionally included in requests for additional tickets
           from the ticket-granting server.  It specifies the
           addresses from which the requested ticket is to be valid.
           Normally it includes the addresses for the client's host.
           If a proxy is requested, this field will contain other
           addresses.  The contents of this field are usually copied
           by the KDC into the caddr field of the resulting ticket.

additional-tickets Additional tickets may be optionally included in a
           request to the ticket-granting server.  If the ENC-TKT-IN-
           SKEY option has been specified, then the session key from
           the additional ticket will be used in place of the server's
           key to encrypt the new ticket.  If more than one option
           which requires additional tickets has been specified, then
           the additional tickets are used in the order specified by
           the ordering of the options bits (see kdc-options, above).

The application code will be either ten (10) or twelve (12) depending
on whether the request is for an initial ticket (AS-REQ) or for an
additional ticket (TGS-REQ).

The optional fields (addresses, authorization-data and additional-
tickets) are only included if necessary to perform the operation
specified in the kdc-options field.

It should be noted that in KRB_TGS_REQ, the protocol version number
appears twice and two different message types appear: the KRB_TGS_REQ
message contains these fields as does the authentication header
(KRB_AP_REQ) that is passed in the padata field.

## 5.4.2. KRB_KDC_REP definition

The KRB_KDC_REP message format is used for the reply from the KDC for
either an initial (AS) request or a subsequent (TGS) request.  There
is no message type for KRB_KDC_REP.  Instead, the type will be either
KRB_AS_REP or KRB_TGS_REP.  The key used to encrypt the ciphertext
part of the reply depends on the message type.  For KRB_AS_REP, the
ciphertext is encrypted in the client's secret key, and the client's
key version number is included in the key version number for the
encrypted data.  For KRB_TGS_REP, the ciphertext is encrypted in the
sub-session key from the Authenticator, or if absent, the session key
from the ticket-granting ticket used in the request.  In that case,
no version number will be present in the EncryptedData sequence.

The KRB_KDC_REP message contains the following fields:

```
AS-REP  ::=     [APPLICATION 11] KDC-REP
TGS-REP ::=     [APPLICATION 13] KDC-REP

KDC-REP ::=     SEQUENCE {
                pvno[0]                     INTEGER,
                msg-type[1]                 INTEGER,
                padata[2]                   SEQUENCE OF PA-DATA OPTIONAL,
                crealm[3]                   Realm,
                cname[4]                    PrincipalName,
                ticket[5]                   Ticket,
                enc-part[6]                 EncryptedData
}

EncASRepPart ::=    [APPLICATION 25[25]] EncKDCRepPart
EncTGSRepPart ::=   [APPLICATION 26] EncKDCRepPart

EncKDCRepPart ::=   SEQUENCE {
                key[0]                      EncryptionKey,
                last-req[1]                 LastReq,
```

```
                    nonce[2]                        INTEGER,
                    key-expiration[3]               KerberosTime OPTIONAL,
                    flags[4]                        TicketFlags,
                    authtime[5]                     KerberosTime,
                    starttime[6]                    KerberosTime OPTIONAL,
                    endtime[7]                      KerberosTime,
                    renew-till[8]                   KerberosTime OPTIONAL,
                    srealm[9]                       Realm,
                    sname[10]                       PrincipalName,
                    caddr[11]                       HostAddresses OPTIONAL
            }
```

   NOTE: In EncASRepPart, the application code in the encrypted
         part of a message provides an additional check that
         the message was decrypted properly.

   pvno and msg-type These fields are described above in section 5.4.1.
            msg-type is either KRB_AS_REP or KRB_TGS_REP.

   padata   This field is described in detail in section 5.4.1. One
              possible use for this field is to encode an alternate
              "mix-in" string to be used with a string-to-key algorithm
              (such as is described in section 6.3.2). This ability is
              useful to ease transitions if a realm name needs to change
              (e.g., when a company is acquired); in such a case all
              existing password-derived entries in the KDC database would
              be flagged as needing a special mix-in string until the
              next password change.

   crealm, cname, srealm and sname These fields are the same as those
            described for the ticket in section 5.3.1.

   ticket   The newly-issued ticket, from section 5.3.1.

   enc-part   This field is a place holder for the ciphertext and related
              information that forms the encrypted part of a message.
              The description of the encrypted part of the message
              follows each appearance of this field. The encrypted part
              is encoded as described in section 6.1.

   key   This field is the same as described for the ticket in
              section 5.3.1.

   last-req   This field is returned by the KDC and specifies the time(s)
              of the last request by a principal. Depending on what
              information is available, this might be the last time that
              a request for a ticket-granting ticket was made, or the
              last time that a request based on a ticket-granting ticket

was successful. It also might cover all servers for a
realm, or just the particular server. Some implementations
may display this information to the user to aid in
discovering unauthorized use of one's identity. It is
similar in spirit to the last login time displayed when
logging into timesharing systems.

nonce   This field is described above in section 5.4.1.

key-expiration The key-expiration field is part of the response from
the KDC and specifies the time that the client's secret key
is due to expire. The expiration might be the result of
password aging or an account expiration. This field will
usually be left out of the TGS reply since the response to
the TGS request is encrypted in a session key and no client
information need be retrieved from the KDC database. It is
up to the application client (usually the login program) to
take appropriate action (such as notifying the user) if the
expira   tion time is imminent.

flags, authtime, starttime, endtime, renew-till and caddr These
fields are duplicates of those found in the encrypted
portion of the attached ticket (see section 5.3.1),
provided so the client may verify they match the intended
request and to assist in proper ticket caching. If the
message is of type KRB_TGS_REP, the caddr field will only
be filled in if the request was for a proxy or forwarded
ticket, or if the user is substituting a subset of the
addresses from the ticket granting ticket. If the client-
requested addresses are not present or not used, then the
addresses contained in the ticket will be the same as those
included in the ticket-granting ticket.

5.5. Client/Server (CS) message specifications

   This section specifies the format of the messages used for the
   authentication of the client to the application server.

5.5.1. KRB_AP_REQ definition

   The KRB_AP_REQ message contains the Kerberos protocol version number,
   the message type KRB_AP_REQ, an options field to indicate any options
   in use, and the ticket and authenticator themselves. The KRB_AP_REQ
   message is often referred to as the "authentication header".

   AP-REQ ::=   [APPLICATION 14] SEQUENCE {
                     pvno[0]   INTEGER,
                     msg-type[1]   INTEGER,

```
                  ap-options[2]                   APOptions,
                  ticket[3]                       Ticket,
                  authenticator[4]                EncryptedData
   }

   APOptions ::=    BIT STRING {
                  reserved(0),
                  use-session-key(1),
                  mutual-required(2)
   }
```

   pvno and msg-type These fields are described above in section 5.4.1.
            msg-type is KRB_AP_REQ.

   ap-options This field appears in the application request (KRB_AP_REQ)
            and affects the way the request is processed. It is a
            bit-field, where the selected options are indicated by the
            bit being set (1), and the unselected options and reserved
            fields being reset (0). The encoding of the bits is
            specified in section 5.2. The meanings of the options are:

            Bit(s) Name   Description

            0   RESERVED   Reserved for future expansion of
                                    this field.

            1   USE-SESSION-KEYThe USE-SESSION-KEY option indicates
                                    that the ticket the client is
                                    presenting to a server is encrypted in
                                    the session key from the server's
                                    ticket-granting ticket. When this
                                    option is not specified, the ticket is
                                    encrypted in the server's secret key.

            2   MUTUAL-REQUIREDThe MUTUAL-REQUIRED option tells the
                                    server that the client requires mutual
                                    authentication, and that it must
                                    respond with a KRB_AP_REP message.

            3-31   RESERVED   Reserved for future use.

   ticket   This field is a ticket authenticating the client to the
            server.

   authenticator This contains the authenticator, which includes the
            client's choice of a subkey. Its encoding is described in
            section 5.3.2.

5.5.2. KRB_AP_REP definition

   The KRB_AP_REP message contains the Kerberos protocol version number,
   the message type, and an encrypted timestamp. The message is sent in
   in response to an application request (KRB_AP_REQ) where the mutual
   authentication option has been selected in the ap-options field.

```
   AP-REP ::=          [APPLICATION 15] SEQUENCE {
           pvno[0]                     INTEGER,
           msg-type[1]                 INTEGER,
           enc-part[2]                 EncryptedData
   }

   EncAPRepPart ::=   [APPLICATION 27]    SEQUENCE {
           ctime[0]                    KerberosTime,
           cusec[1]                    INTEGER,
           subkey[2]                   EncryptionKey OPTIONAL,
           seq-number[3]               INTEGER OPTIONAL
   }
```

   NOTE: in EncAPRepPart, the application code in the encrypted part of
   a message provides an additional check that the message was decrypted
   properly.

   The encoded EncAPRepPart is encrypted in the shared session key of
   the ticket. The optional subkey field can be used in an
   application-arranged negotiation to choose a per association session
   key.

   pvno and msg-type These fields are described above in section 5.4.1.
           msg-type is KRB_AP_REP.

   enc-part This field is described above in section 5.4.2.

   ctime   This field contains the current time on the client's host.

   cusec   This field contains the microsecond part of the client's
             timestamp.

   subkey   This field contains an encryption key which is to be used
             to protect this specific application session. See section
             3.2.6 for specifics on how this field is used to negotiate
             a key. Unless an application specifies otherwise, if this
             field is left out, the sub-session key from the
             authenticator, or if also left out, the session key from
             the ticket will be used.

5.5.3. Error message reply

   If an error occurs while processing the application request, the
   KRB_ERROR message will be sent in response. See section 5.9.1 for
   the format of the error message. The cname and crealm fields may be
   left out if the server cannot determine their appropriate values from
   the corresponding KRB_AP_REQ message. If the authenticator was
   decipherable, the ctime and cusec fields will contain the values from
   it.

5.6. KRB_SAFE message specification

   This section specifies the format of a message that can be used by
   either side (client or server) of an application to send a tamper-
   proof message to its peer. It presumes that a session key has
   previously been exchanged (for example, by using the
   KRB_AP_REQ/KRB_AP_REP messages).

5.6.1. KRB_SAFE definition

   The KRB_SAFE message contains user data along with a collision-proof
   checksum keyed with the session key. The message fields are:

   KRB-SAFE ::=   [APPLICATION 20] SEQUENCE {
               pvno[0]   INTEGER,
               msg-type[1]   INTEGER,
               safe-body[2]   KRB-SAFE-BODY,
               cksum[3]   Checksum
   }

   KRB-SAFE-BODY ::=   SEQUENCE {
               user-data[0]   OCTET STRING,
               timestamp[1]   KerberosTime OPTIONAL,
               usec[2]   INTEGER OPTIONAL,
               seq-number[3]   INTEGER OPTIONAL,
               s-address[4]   HostAddress,
               r-address[5]   HostAddress OPTIONAL
   }

   pvno and msg-type These fields are described above in section 5.4.1.
            msg-type is KRB_SAFE.

   safe-body This field is a placeholder for the body of the KRB-SAFE
            message. It is to be encoded separately and then have the
            checksum computed over it, for use in the cksum field.

   cksum   This field contains the checksum of the application data.
            Checksum details are described in section 6.4. The

checksum is computed over the encoding of the KRB-SAFE-BODY
sequence.

user-data This field is part of the KRB_SAFE and KRB_PRIV messages
          and contain the application specific data that is being
          passed from the sender to the recipient.

timestamp This field is part of the KRB_SAFE and KRB_PRIV messages.
          Its contents are the current time as known by the sender of
          the message. By checking the timestamp, the recipient of
          the message is able to make sure that it was recently
          generated, and is not a replay.

usec   This field is part of the KRB_SAFE and KRB_PRIV headers.
          It contains the microsecond part of the timestamp.

seq-number This field is described above in section 5.3.2.

s-address This field specifies the address in use by the sender of
          the message.

r-address This field specifies the address in use by the recipient of
          the message. It may be omitted for some uses (such as
          broadcast protocols), but the recipient may arbitrarily
          reject such messages. This field along with s-address can
          be used to help detect messages which have been incorrectly
          or maliciously delivered to the wrong recipient.

5.7. KRB_PRIV message specification

   This section specifies the format of a message that can be used by
   either side (client or server) of an application to securely and
   privately send a message to its peer. It presumes that a session key
   has previously been exchanged (for example, by using the
   KRB_AP_REQ/KRB_AP_REP messages).

5.7.1. KRB_PRIV definition

   The KRB_PRIV message contains user data encrypted in the Session Key.
   The message fields are:

   KRB-PRIV ::=   [APPLICATION 21] SEQUENCE {
               pvno[0]   INTEGER,
               msg-type[1]   INTEGER,
               enc-part[3]   EncryptedData
   }

```
EncKrbPrivPart ::=    [APPLICATION 28] SEQUENCE {
            user-data[0]                OCTET STRING,
            timestamp[1]                KerberosTime OPTIONAL,
            usec[2]                     INTEGER OPTIONAL,
            seq-number[3]               INTEGER OPTIONAL,
            s-address[4]                HostAddress, -- sender's addr
            r-address[5]                HostAddress OPTIONAL
                                                    -- recip's addr
}
```

NOTE: In EncKrbPrivPart, the application code in the encrypted part
of a message provides an additional check that the message was
decrypted properly.

pvno and msg-type These fields are described above in section 5.4.1.
          msg-type is KRB_PRIV.

enc-part This field holds an encoding of the EncKrbPrivPart sequence
          encrypted under the session key (If supported by the
          encryption method in use, an initialization vector may be
          passed to the encryption procedure, in order to achieve
          proper cipher chaining. The initialization vector might
          come from the last block of the ciphertext from the
          previous KRB_PRIV message, but it is the application's
          choice whether or not to use such an initialization vector.
          If left out, the default initialization vector for the
          encryption algorithm will be used.). This encrypted
          encoding is used for the enc-part field of the KRB-PRIV
          message. See section 6 for the format of the ciphertext.

user-data, timestamp, usec, s-address and r-address These fields are
          described above in section 5.6.1.

seq-number This field is described above in section 5.3.2.

5.8. KRB_CRED message specification

   This section specifies the format of a message that can be used to
   send Kerberos credentials from one principal to another. It is
   presented here to encourage a common mechanism to be used by
   applications when forwarding tickets or providing proxies to
   subordinate servers. It presumes that a session key has already been
   exchanged perhaps by using the KRB_AP_REQ/KRB_AP_REP messages.

5.8.1. KRB_CRED definition

   The KRB_CRED message contains a sequence of tickets to be sent and
   information needed to use the tickets, including the session key from

each. The information needed to use the tickets is encryped under an
encryption key previously exchanged. The message fields are:

```
KRB-CRED            ::= [APPLICATION 22]   SEQUENCE {
                    pvno[0]                INTEGER,
                    msg-type[1]            INTEGER, -- KRB_CRED
                    tickets[2]             SEQUENCE OF Ticket,
                    enc-part[3]            EncryptedData
}

EncKrbCredPart      ::= [APPLICATION 29]   SEQUENCE {
                    ticket-info[0]         SEQUENCE OF KrbCredInfo,
                    nonce[1]               INTEGER OPTIONAL,
                    timestamp[2]           KerberosTime OPTIONAL,
                    usec[3]                INTEGER OPTIONAL,
                    s-address[4]           HostAddress OPTIONAL,
                    r-address[5]           HostAddress OPTIONAL
}

KrbCredInfo         ::=                    SEQUENCE {
                    key[0]                 EncryptionKey,
                    prealm[1]              Realm OPTIONAL,
                    pname[2]               PrincipalName OPTIONAL,
                    flags[3]               TicketFlags OPTIONAL,
                    authtime[4]            KerberosTime OPTIONAL,
                    starttime[5]           KerberosTime OPTIONAL,
                    endtime[6]             KerberosTime OPTIONAL
                    renew-till[7]          KerberosTime OPTIONAL,
                    srealm[8]              Realm OPTIONAL,
                    sname[9]               PrincipalName OPTIONAL,
                    caddr[10]              HostAddresses OPTIONAL
}
```

   pvno and msg-type These fields are described above in section 5.4.1.
           msg-type is KRB_CRED.

   tickets
             These are the tickets obtained from the KDC specifically
           for use by the intended recipient. Successive tickets are
           paired with the corresponding KrbCredInfo sequence from the
           enc-part of the KRB-CRED message.

   enc-part This field holds an encoding of the EncKrbCredPart sequence
           encrypted under the session key shared between the sender
           and the intended recipient. This encrypted encoding is
           used for the enc-part field of the KRB-CRED message. See
           section 6 for the format of the ciphertext.

nonce   If practical, an application may require the inclusion of a
           nonce generated by the recipient of the message. If the
           same value is included as the nonce in the message, it
           provides evidence that the message is fresh and has not
           been replayed by an attacker. A nonce must never be re-
           used; it should be generated randomly by the recipient of
           the message and provided to the sender of the mes sage in
           an application specific manner.

timestamp and usec These fields specify the time that the KRB-CRED
           message was generated. The time is used to provide
           assurance that the message is fresh.

s-address and r-address These fields are described above in section
           5.6.1. They are used optionally to provide additional
           assurance of the integrity of the KRB-CRED message.

key   This field exists in the corresponding ticket passed by the
           KRB-CRED message and is used to pass the session key from
           the sender to the intended recipient. The field's encoding
           is described in section 6.2.

The following fields are optional. If present, they can be
associated with the credentials in the remote ticket file. If left
out, then it is assumed that the recipient of the credentials already
knows their value.

prealm and pname The name and realm of the delegated principal
           identity.

flags, authtime, starttime, endtime, renew-till, srealm, sname,
           and caddr These fields contain the values of the
           corresponding fields from the ticket found in the ticket
           field. Descriptions of the fields are identical to the
           descriptions in the KDC-REP message.

5.9. Error message specification

   This section specifies the format for the KRB_ERROR message. The
   fields included in the message are intended to return as much
   information as possible about an error. It is not expected that all
   the information required by the fields will be available for all
   types of errors. If the appropriate information is not available
   when the message is composed, the corresponding field will be left
   out of the message.

   Note that since the KRB_ERROR message is not protected by any
   encryption, it is quite possible for an intruder to synthesize or

modify such a message. In particular, this means that the client
should not use any fields in this message for security-critical
purposes, such as setting a system clock or generating a fresh
authenticator. The message can be useful, however, for advising a
user on the reason for some failure.

5.9.1. KRB_ERROR definition

   The KRB_ERROR message consists of the following fields:

   KRB-ERROR ::=   [APPLICATION 30] SEQUENCE {
                      pvno[0]   INTEGER,
                      msg-type[1]   INTEGER,
                      ctime[2]   KerberosTime OPTIONAL,
                      cusec[3]   INTEGER OPTIONAL,
                      stime[4]   KerberosTime,
                      susec[5]   INTEGER,
                      error-code[6]   INTEGER,
                      crealm[7]   Realm OPTIONAL,
                      cname[8]   PrincipalName OPTIONAL,
                      realm[9]   Realm, -- Correct realm
                      sname[10]   PrincipalName, -- Correct name
                      e-text[11]   GeneralString OPTIONAL,
                      e-data[12]   OCTET STRING OPTIONAL
   }

   pvno and msg-type These fields are described above in section 5.4.1.
            msg-type is KRB_ERROR.

   ctime   This field is described above in section 5.4.1.

   cusec   This field is described above in section 5.5.2.

   stime   This field contains the current time on the server. It is
              of type KerberosTime.

   susec   This field contains the microsecond part of the server's
              timestamp. Its value ranges from 0 to 999. It appears
              along with stime. The two fields are used in conjunction to
              specify a reasonably accurate timestamp.

   error-code This field contains the error code returned by Kerberos or
              the server when a request fails. To interpret the value of
              this field see the list of error codes in section 8.
              Implementations are encouraged to provide for national
              language support in the display of error messages.

   crealm, cname, srealm and sname These fields are described above in

section 5.3.1.

e-text   This field contains additional text to help explain the
           error code associated with the failed request (for example,
           it might include a principal name which was unknown).

e-data   This field contains additional data about the error for use
           by the application to help it recover from or handle the
           error. If the errorcode is KDC_ERR_PREAUTH_REQUIRED, then
           the e-data field will contain an encoding of a sequence of
           padata fields, each corresponding to an acceptable pre-
           authentication method and optionally containing data for
           the method:

    METHOD-DATA ::=   SEQUENCE of PA-DATA

If the error-code is KRB_AP_ERR_METHOD, then the e-data field will
contain an encoding of the following sequence:

    METHOD-DATA ::=   SEQUENCE {
                        method-type[0]   INTEGER,
                        method-data[1]   OCTET STRING OPTIONAL
       }

method-type will indicate the required alternate method; method-data
will contain any required additional information.

6. Encryption and Checksum Specifications

   The Kerberos protocols described in this document are designed to use
   stream encryption ciphers, which can be simulated using commonly
   available block encryption ciphers, such as the Data Encryption
   Standard [11], in conjunction with block chaining and checksum
   methods [12]. Encryption is used to prove the identities of the
   network entities participating in message exchanges. The Key
   Distribution Center for each realm is trusted by all principals
   registered in that realm to store a secret key in confidence. Proof
   of knowledge of this secret key is used to verify the authenticity of
   a principal.

   The KDC uses the principal's secret key (in the AS exchange) or a
   shared session key (in the TGS exchange) to encrypt responses to
   ticket requests; the ability to obtain the secret key or session key
   implies the knowledge of the appropriate keys and the identity of the
   KDC. The ability of a principal to decrypt the KDC response and
   present a Ticket and a properly formed Authenticator (generated with
   the session key from the KDC response) to a service verifies the
   identity of the principal; likewise the ability of the service to

extract the session key from the Ticket and prove its knowledge
thereof in a response verifies the identity of the service.

The Kerberos protocols generally assume that the encryption used is
secure from cryptanalysis; however, in some cases, the order of
fields in the encrypted portions of messages are arranged to minimize
the effects of poorly chosen keys. It is still important to choose
good keys. If keys are derived from user-typed passwords, those
passwords need to be well chosen to make brute force attacks more
difficult. Poorly chosen keys still make easy targets for intruders.

The following sections specify the encryption and checksum mechanisms
currently defined for Kerberos. The encodings, chaining, and padding
requirements for each are described. For encryption methods, it is
often desirable to place random information (often referred to as a
confounder) at the start of the message. The requirements for a
confounder are specified with each encryption mechanism.

Some encryption systems use a block-chaining method to improve the
the security characteristics of the ciphertext. However, these
chaining methods often don't provide an integrity check upon
decryption. Such systems (such as DES in CBC mode) must be augmented
with a checksum of the plaintext which can be verified at decryption
and used to detect any tampering or damage. Such checksums should be
good at detecting burst errors in the input. If any damage is
detected, the decryption routine is expected to return an error
indicating the failure of an integrity check. Each encryption type is
expected to provide and verify an appropriate checksum. The
specification of each encryption method sets out its checksum
requirements.

Finally, where a key is to be derived from a user's password, an
algorithm for converting the password to a key of the appropriate
type is included. It is desirable for the string to key function to
be one-way, and for the mapping to be different in different realms.
This is important because users who are registered in more than one
realm will often use the same password in each, and it is desirable
that an attacker compromising the Kerberos server in one realm not
obtain or derive the user's key in another.

For a discussion of the integrity characteristics of the candidate
encryption and checksum methods considered for Kerberos, the the
reader is referred to [13].

6.1. Encryption Specifications

The following ASN.1 definition describes all encrypted messages. The
enc-part field which appears in the unencrypted part of messages in

section 5 is a sequence consisting of an encryption type, an optional
key version number, and the ciphertext.

```
EncryptedData ::=       SEQUENCE {
                        etype[0]        INTEGER, -- EncryptionType
                        kvno[1]         INTEGER OPTIONAL,
                        cipher[2]       OCTET STRING -- ciphertext
}
```

etype   This field identifies which encryption algorithm was used
          to encipher the cipher. Detailed specifications for
          selected encryption types appear later in this section.

kvno   This field contains the version number of the key under
          which data is encrypted. It is only present in messages
          encrypted under long lasting keys, such as principals'
          secret keys.

cipher   This field contains the enciphered text, encoded as an
          OCTET STRING.

The cipher field is generated by applying the specified encryption
algorithm to data composed of the message and algorithm-specific
inputs. Encryption mechanisms defined for use with Kerberos must
take sufficient measures to guarantee the integrity of the plaintext,
and we recommend they also take measures to protect against
precomputed dictionary attacks. If the encryption algorithm is not
itself capable of doing so, the protections can often be enhanced by
adding a checksum and a confounder.

The suggested format for the data to be encrypted includes a
confounder, a checksum, the encoded plaintext, and any necessary
padding. The msg-seq field contains the part of the protocol message
described in section 5 which is to be encrypted. The confounder,
checksum, and padding are all untagged and untyped, and their length
is exactly sufficient to hold the appropriate item. The type and
length is implicit and specified by the particular encryption type
being used (etype). The format for the data to be encrypted is
described in the following diagram:

```
    +-----------+----------+-------------+-----+
    |confounder |   check  |   msg-seq   | pad |
    +-----------+----------+-------------+-----+
```

The format cannot be described in ASN.1, but for those who prefer an
ASN.1-like notation:

```
CipherText ::=    ENCRYPTED         SEQUENCE {
        confounder[0]    UNTAGGED OCTET STRING(conf_length)      OPTIONAL,
        check[1]         UNTAGGED OCTET STRING(checksum_length) OPTIONAL,
        msg-seq[2]       MsgSequence,
        pad              UNTAGGED OCTET STRING(pad_length) OPTIONAL
}
```

In the above specification, UNTAGGED OCTET STRING(length) is the
notation for an octet string with its tag and length removed. It is
not a valid ASN.1 type. The tag bits and length must be removed from
the confounder since the purpose of the confounder is so that the
message starts with random data, but the tag and its length are
fixed. For other fields, the length and tag would be redundant if
they were included because they are specified by the encryption type.

One generates a random confounder of the appropriate length, placing
it in confounder; zeroes out check; calculates the appropriate
checksum over confounder, check, and msg-seq, placing the result in
check; adds the necessary padding; then encrypts using the specified
encryption type and the appropriate key.

Unless otherwise specified, a definition of an encryption algorithm
that specifies a checksum, a length for the confounder field, or an
octet boundary for padding uses this ciphertext format (The ordering
of the fields in the CipherText is important. Additionally, messages
encoded in this format must include a length as part of the msg-seq
field. This allows the recipient to verify that the message has not
been truncated. Without a length, an attacker could use a chosen
plaintext attack to generate a message which could be truncated,
while leaving the checksum intact. Note that if the msg-seq is an
encoding of an ASN.1 SEQUENCE or OCTET STRING, then the length is
part of that encoding.). Those fields which are not specified will be
omitted.

In the interest of allowing all implementations using a particular
encryption type to communicate with all others using that type, the
specification of an encryption type defines any checksum that is
needed as part of the encryption process. If an alternative checksum
is to be used, a new encryption type must be defined.

Some cryptosystems require additional information beyond the key and
the data to be encrypted. For example, DES, when used in cipher-
block-chaining mode, requires an initialization vector. If required,
the description for each encryption type must specify the source of
such additional information.

6.2. Encryption Keys

   The sequence below shows the encoding of an encryption key:

         EncryptionKey ::=   SEQUENCE {
                              keytype[0]   INTEGER,
                              keyvalue[1]   OCTET STRING
         }

   keytype   This field specifies the type of encryption key that
             follows in the keyvalue field. It will almost always
             correspond to the encryption algorithm used to generate the
             EncryptedData, though more than one algorithm may use the
             same type of key (the mapping is many to one). This might
             happen, for example, if the encryption algorithm uses an
             alternate checksum algorithm for an integrity check, or a
             different chaining mechanism.

   keyvalue This field contains the key itself, encoded as an octet
             string.

   All negative values for the encryption key type are reserved for
   local use. All non-negative values are reserved for officially
   assigned type fields and interpretations.

6.3. Encryption Systems

6.3.1. The NULL Encryption System (null)

   If no encryption is in use, the encryption system is said to be the
   NULL encryption system. In the NULL encryption system there is no
   checksum, confounder or padding. The ciphertext is simply the
   plaintext. The NULL Key is used by the null encryption system and is
   zero octets in length, with keytype zero (0).

6.3.2. DES in CBC mode with a CRC-32 checksum (des-cbc-crc)

   The des-cbc-crc encryption mode encrypts information under the Data
   Encryption Standard [11] using the cipher block chaining mode [12].
   A CRC-32 checksum (described in ISO 3309 [14]) is applied to the
   confounder and message sequence (msg-seq) and placed in the cksum
   field. DES blocks are 8 bytes. As a result, the data to be
   encrypted (the concatenation of confounder, checksum, and message)
   must be padded to an 8 byte boundary before encryption. The details
   of the encryption of this data are identical to those for the des-
   cbc-md5 encryption mode.

   Note that, since the CRC-32 checksum is not collisionproof, an

attacker could use a probabilistic chosenplaintext attack to generate
a valid message even if a confounder is used [13]. The use of
collision-proof checksums is recommended for environments where such
attacks represent a significant threat. The use of the CRC-32 as the
checksum for ticket or authenticator is no longer mandated as an
interoperability requirement for Kerberos Version 5 Specification 1
(See section 9.1 for specific details).

6.3.3. DES in CBC mode with an MD4 checksum (des-cbc-md4)

The des-cbc-md4 encryption mode encrypts information under the Data
Encryption Standard [11] using the cipher block chaining mode [12].
An MD4 checksum (described in [15]) is applied to the confounder and
message sequence (msg-seq) and placed in the cksum field. DES blocks
are 8 bytes. As a result, the data to be encrypted (the
concatenation of confounder, checksum, and message) must be padded to
an 8 byte boundary before encryption. The details of the encryption
of this data are identical to those for the descbc-md5 encryption
mode.

6.3.4. DES in CBC mode with an MD5 checksum (des-cbc-md5)

The des-cbc-md5 encryption mode encrypts information under the Data
Encryption Standard [11] using the cipher block chaining mode [12].
An MD5 checksum (described in [16]) is applied to the confounder and
message sequence (msg-seq) and placed in the cksum field. DES blocks
are 8 bytes. As a result, the data to be encrypted (the
concatenation of confounder, checksum, and message) must be padded to
an 8 byte boundary before encryption.

Plaintext and DES ciphtertext are encoded as 8-octet blocks which are
concatenated to make the 64-bit inputs for the DES algorithms. The
first octet supplies the 8 most significant bits (with the octet's
MSbit used as the DES input block's MSbit, etc.), the second octet
the next 8 bits, ..., and the eighth octet supplies the 8 least
significant bits.

Encryption under DES using cipher block chaining requires an
additional input in the form of an initialization vector. Unless
otherwise specified, zero should be used as the initialization
vector. Kerberos' use of DES requires an 8-octet confounder.

The DES specifications identify some "weak" and "semiweak" keys;
those keys shall not be used for encrypting messages for use in
Kerberos. Additionally, because of the way that keys are derived for
the encryption of checksums, keys shall not be used that yield "weak"
or "semi-weak" keys when eXclusive-ORed with the constant
F0F0F0F0F0F0F0F0.

A DES key is 8 octets of data, with keytype one (1). This consists
of 56 bits of key, and 8 parity bits (one per octet). The key is
encoded as a series of 8 octets written in MSB-first order. The bits
within the key are also encoded in MSB order. For example, if the
encryption key is:
(B1,B2,...,B7,P1,B8,...,B14,P2,B15,...,B49,P7,B50,...,B56,P8) where
B1,B2,...,B56 are the key bits in MSB order, and P1,P2,...,P8 are the
parity bits, the first octet of the key would be B1,B2,...,B7,P1
(with B1 as the MSbit). [See the FIPS 81 introduction for
reference.]

To generate a DES key from a text string (password), the text string
normally must have the realm and each component of the principal's
name appended(In some cases, it may be necessary to use a different
"mix-in" string for compatibility reasons; see the discussion of
padata in section 5.4.2.), then padded with ASCII nulls to an 8 byte
boundary. This string is then fan-folded and eXclusive-ORed with
itself to form an 8 byte DES key. The parity is corrected on the
key, and it is used to generate a DES CBC checksum on the initial
string (with the realm and name appended). Next, parity is corrected
on the CBC checksum. If the result matches a "weak" or "semiweak"
key as described in the DES specification, it is eXclusive-ORed with
the constant 00000000000000F0. Finally, the result is returned as
the key. Pseudocode follows:

```
    string_to_key(string,realm,name) {
        odd = 1;
        s = string + realm;
        for(each component in name) {
            s = s + component;
        }
        tempkey = NULL;
        pad(s); /* with nulls to 8 byte boundary */
        for(8byteblock in s) {
            if(odd == 0)  {
                odd = 1;
                reverse(8byteblock)
            }
            else odd = 0;
            tempkey = tempkey XOR 8byteblock;
        }
        fixparity(tempkey);
        key = DES-CBC-check(s,tempkey);
        fixparity(key);
        if(is_weak_key_key(key))
            key = key XOR 0xF0;
        return(key);
    }
```

6.4. Checksums

   The following is the ASN.1 definition used for a checksum:

           Checksum ::=   SEQUENCE {
                             cksumtype[0]   INTEGER,
                             checksum[1]   OCTET STRING
           }

   cksumtype This field indicates the algorithm used to generate the
           accompanying checksum.

   checksum This field contains the checksum itself, encoded
           as an octet string.

   Detailed specification of selected checksum types appear later in
   this section. Negative values for the checksum type are reserved for
   local use. All non-negative values are reserved for officially
   assigned type fields and interpretations.

   Checksums used by Kerberos can be classified by two properties:
   whether they are collision-proof, and whether they are keyed. It is
   infeasible to find two plaintexts which generate the same checksum
   value for a collision-proof checksum. A key is required to perturb
   or initialize the algorithm in a keyed checksum. To prevent
   message-stream modification by an active attacker, unkeyed checksums
   should only be used when the checksum and message will be
   subsequently encrypted (e.g., the checksums defined as part of the
   encryption algorithms covered earlier in this section). Collision-
   proof checksums can be made tamper-proof as well if the checksum
   value is encrypted before inclusion in a message. In such cases, the
   composition of the checksum and the encryption algorithm must be
   considered a separate checksum algorithm (e.g., RSA-MD5 encrypted
   using DES is a new checksum algorithm of type RSA-MD5-DES). For most
   keyed checksums, as well as for the encrypted forms of collisionproof
   checksums, Kerberos prepends a confounder before the checksum is
   calculated.

6.4.1. The CRC-32 Checksum (crc32)

   The CRC-32 checksum calculates a checksum based on a cyclic
   redundancy check as described in ISO 3309 [14]. The resulting
   checksum is four (4) octets in length. The CRC-32 is neither keyed
   nor collision-proof. The use of this checksum is not recommended.
   An attacker using a probabilistic chosen-plaintext attack as
   described in [13] might be able to generate an alternative message
   that satisfies the checksum. The use of collision-proof checksums is
   recommended for environments where such attacks represent a

significant threat.

6.4.2. The RSA MD4 Checksum (rsa-md4)

The RSA-MD4 checksum calculates a checksum using the RSA MD4
algorithm [15]. The algorithm takes as input an input message of
arbitrary length and produces as output a 128-bit (16 octet)
checksum. RSA-MD4 is believed to be collision-proof.

6.4.3. RSA MD4 Cryptographic Checksum Using DES (rsa-md4des)

The RSA-MD4-DES checksum calculates a keyed collisionproof checksum
by prepending an 8 octet confounder before the text, applying the RSA
MD4 checksum algorithm, and encrypting the confounder and the
checksum using DES in cipher-block-chaining (CBC) mode using a
variant of the key, where the variant is computed by eXclusive-ORing
the key with the constant F0F0F0F0F0F0F0F0 (A variant of the key is
used to limit the use of a key to a particular function, separating
the functions of generating a checksum from other encryption
performed using the session key. The constant F0F0F0F0F0F0F0F0 was
chosen because it maintains key parity. The properties of DES
precluded the use of the complement. The same constant is used for
similar purpose in the Message Integrity Check in the Privacy
Enhanced Mail standard.). The initialization vector should be zero.
The resulting checksum is 24 octets long (8 octets of which are
redundant). This checksum is tamper-proof and believed to be
collision-proof.

The DES specifications identify some "weak keys"; those keys shall
not be used for generating RSA-MD4 checksums for use in Kerberos.

The format for the checksum is described in the following diagram:

```
    +--+--+--+--+--+--+--+--
    |  des-cbc(confounder
    +--+--+--+--+--+--+--+--
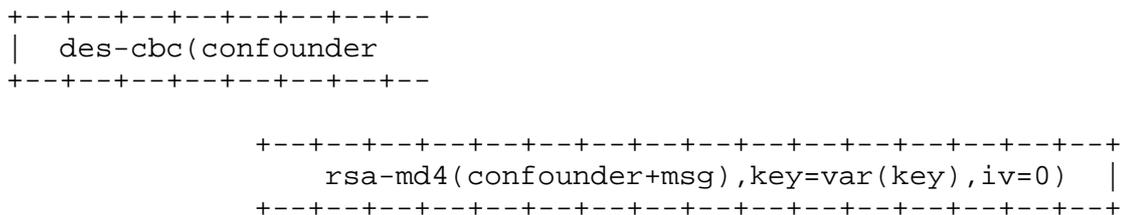
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                rsa-md4(confounder+msg),key=var(key),iv=0)  |
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The format cannot be described in ASN.1, but for those who prefer an
ASN.1-like notation:

```
rsa-md4-des-checksum ::=    ENCRYPTED         UNTAGGED SEQUENCE {
                            confounder[0]   UNTAGGED OCTET STRING(8),
                            check[1]        UNTAGGED OCTET STRING(16)
}
```

6.4.4. The RSA MD5 Checksum (rsa-md5)

   The RSA-MD5 checksum calculates a checksum using the RSA MD5
   algorithm [16]. The algorithm takes as input an input message of
   arbitrary length and produces as output a 128-bit (16 octet)
   checksum. RSA-MD5 is believed to be collision-proof.

6.4.5. RSA MD5 Cryptographic Checksum Using DES (rsa-md5des)

   The RSA-MD5-DES checksum calculates a keyed collisionproof checksum
   by prepending an 8 octet confounder before the text, applying the RSA
   MD5 checksum algorithm, and encrypting the confounder and the
   checksum using DES in cipher-block-chaining (CBC) mode using a
   variant of the key, where the variant is computed by eXclusive-ORing
   the key with the constant F0F0F0F0F0F0F0F0. The initialization
   vector should be zero. The resulting checksum is 24 octets long (8
   octets of which are redundant). This checksum is tamper-proof and
   believed to be collision-proof.

   The DES specifications identify some "weak keys"; those keys shall
   not be used for encrypting RSA-MD5 checksums for use in Kerberos.

   The format for the checksum is described in the following diagram:

```
      +--+--+--+--+--+--+--+--
      |   des-cbc(confounder
      +--+--+--+--+--+--+--+--
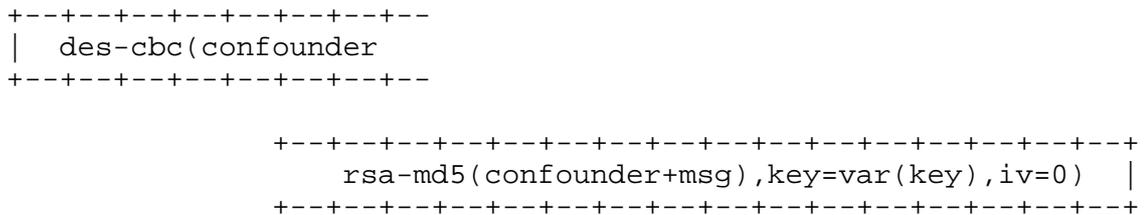
                     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                        rsa-md5(confounder+msg),key=var(key),iv=0)   |
                     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

   The format cannot be described in ASN.1, but for those who prefer an
   ASN.1-like notation:

```
   rsa-md5-des-checksum ::=    ENCRYPTED       UNTAGGED SEQUENCE {
                               confounder[0]   UNTAGGED OCTET STRING(8),
                               check[1]        UNTAGGED OCTET STRING(16)
   }
```

6.4.6. DES cipher-block chained checksum (des-mac)

   The DES-MAC checksum is computed by prepending an 8 octet confounder
   to the plaintext, performing a DES CBC-mode encryption on the result
   using the key and an initialization vector of zero, taking the last
   block of the ciphertext, prepending the same confounder and
   encrypting the pair using DES in cipher-block-chaining (CBC) mode
   using a a variant of the key, where the variant is computed by

eXclusive-ORing the key with the constant F0F0F0F0F0F0F0F0. The
initialization vector should be zero. The resulting checksum is 128
bits (16 octets) long, 64 bits of which are redundant. This checksum
is tamper-proof and collision-proof.

The format for the checksum is described in the following diagram:

```
    +--+--+--+--+--+--+--+--+--
    |   des-cbc(confounder
    +--+--+--+--+--+--+--+--+--
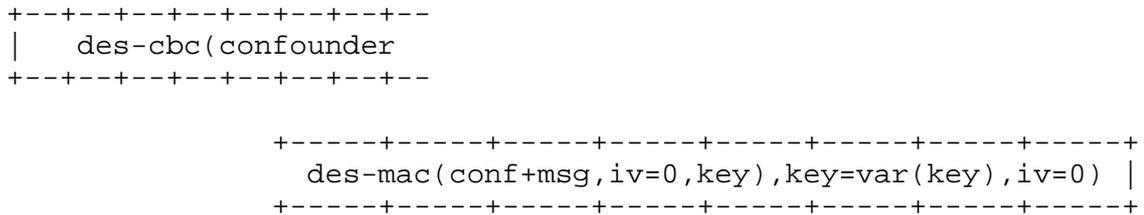
                   +-----+-----+-----+-----+-----+-----+-----+-----+
                     des-mac(conf+msg,iv=0,key),key=var(key),iv=0) |
                   +-----+-----+-----+-----+-----+-----+-----+-----+
```

The format cannot be described in ASN.1, but for those who prefer an
ASN.1-like notation:

```
des-mac-checksum ::=      ENCRYPTED      UNTAGGED SEQUENCE {
                          confounder[0]  UNTAGGED OCTET STRING(8),
                          check[1]       UNTAGGED OCTET STRING(8)
}
```

The DES specifications identify some "weak" and "semiweak" keys;
those keys shall not be used for generating DES-MAC checksums for use
in Kerberos, nor shall a key be used whose veriant is "weak" or
"semi-weak".

6.4.7. RSA MD4 Cryptographic Checksum Using DES alternative
       (rsa-md4-des-k)

The RSA-MD4-DES-K checksum calculates a keyed collision-proof
checksum by applying the RSA MD4 checksum algorithm and encrypting
the results using DES in cipherblock-chaining (CBC) mode using a DES
key as both key and initialization vector. The resulting checksum is
16 octets long. This checksum is tamper-proof and believed to be
collision-proof. Note that this checksum type is the old method for
encoding the RSA-MD4-DES checksum and it is no longer recommended.

6.4.8. DES cipher-block chained checksum alternative (desmac-k)

The DES-MAC-K checksum is computed by performing a DES CBC-mode
encryption of the plaintext, and using the last block of the
ciphertext as the checksum value. It is keyed with an encryption key
and an initialization vector; any uses which do not specify an
additional initialization vector will use the key as both key and
initialization vector. The resulting checksum is 64 bits (8 octets)
long. This checksum is tamper-proof and collision-proof. Note that

this checksum type is the old method for encoding the DESMAC checksum
and it is no longer recommended.

The DES specifications identify some "weak keys"; those keys shall
not be used for generating DES-MAC checksums for use in Kerberos.

7. Naming Constraints

7.1. Realm Names

Although realm names are encoded as GeneralStrings and although a
realm can technically select any name it chooses, interoperability
across realm boundaries requires agreement on how realm names are to
be assigned, and what information they imply.

To enforce these conventions, each realm must conform to the
conventions itself, and it must require that any realms with which
inter-realm keys are shared also conform to the conventions and
require the same from its neighbors.

There are presently four styles of realm names: domain, X500, other,
and reserved. Examples of each style follow:

         domain: host.subdomain.domain (example)
          X500:   C=US/O=OSF (example)
          other: NAMETYPE:rest/of.name=without-restrictions (example)
       reserved: reserved, but will not conflict with above

Domain names must look like domain names: they consist of components
separated by periods (.) and they contain neither colons (:) nor
slashes (/).

X.500 names contain an equal (=) and cannot contain a colon (:)
before the equal. The realm names for X.500 names will be string
representations of the names with components separated by slashes.
Leading and trailing slashes will not be included.

Names that fall into the other category must begin with a prefix that
contains no equal (=) or period (.) and the prefix must be followed
by a colon (:) and the rest of the name. All prefixes must be
assigned before they may be used. Presently none are assigned.

The reserved category includes strings which do not fall into the
first three categories. All names in this category are reserved. It
is unlikely that names will be assigned to this category unless there
is a very strong argument for not using the "other" category.

These rules guarantee that there will be no conflicts between the

various name styles. The following additional constraints apply to
the assignment of realm names in the domain and X.500 categories: the
name of a realm for the domain or X.500 formats must either be used
by the organization owning (to whom it was assigned) an Internet
domain name or X.500 name, or in the case that no such names are
registered, authority to use a realm name may be derived from the
authority of the parent realm. For example, if there is no domain
name for E40.MIT.EDU, then the administrator of the MIT.EDU realm can
authorize the creation of a realm with that name.

This is acceptable because the organization to which the parent is
assigned is presumably the organization authorized to assign names to
its children in the X.500 and domain name systems as well. If the
parent assigns a realm name without also registering it in the domain
name or X.500 hierarchy, it is the parent's responsibility to make
sure that there will not in the future exists a name identical to the
realm name of the child unless it is assigned to the same entity as
the realm name.

7.2. Principal Names

As was the case for realm names, conventions are needed to ensure
that all agree on what information is implied by a principal name.
The name-type field that is part of the principal name indicates the
kind of information implied by the name. The name-type should be
treated as a hint. Ignoring the name type, no two names can be the
same (i.e., at least one of the components, or the realm, must be
different). This constraint may be eliminated in the future. The
following name types are defined:

```
name-type         value    meaning
NT-UNKNOWN          0       Name type not known
NT-PRINCIPAL        1       Just the name of the principal as in
                            DCE, or for users
NT-SRV-INST         2       Service and other unique instance (krbtgt)
NT-SRV-HST          3       Service with host name as instance
                            (telnet, rcommands)
NT-SRV-XHST         4       Service with host as remaining components
NT-UID              5       Unique ID
```

When a name implies no information other than its uniqueness at a
particular time the name type PRINCIPAL should be used. The
principal name type should be used for users, and it might also be
used for a unique server. If the name is a unique machine generated
ID that is guaranteed never to be reassigned then the name type of
UID should be used (note that it is generally a bad idea to reassign
names of any type since stale entries might remain in access control
lists).

If the first component of a name identifies a service and the
remaining components identify an instance of the service in a server
specified manner, then the name type of SRV-INST should be used. An
example of this name type is the Kerberos ticket-granting ticket
which has a first component of krbtgt and a second component
identifying the realm for which the ticket is valid.

If instance is a single component following the service name and the
instance identifies the host on which the server is running, then the
name type SRV-HST should be used. This type is typically used for
Internet services such as telnet and the Berkeley R commands. If the
separate components of the host name appear as successive components
following the name of the service, then the name type SRVXHST should
be used. This type might be used to identify servers on hosts with
X.500 names where the slash (/) might otherwise be ambiguous.

A name type of UNKNOWN should be used when the form of the name is
not known. When comparing names, a name of type UNKNOWN will match
principals authenticated with names of any type. A principal
authenticated with a name of type UNKNOWN, however, will only match
other names of type UNKNOWN.

Names of any type with an initial component of "krbtgt" are reserved
for the Kerberos ticket granting service. See section 8.2.3 for the
form of such names.

7.2.1. Name of server principals

The principal identifier for a server on a host will generally be
composed of two parts: (1) the realm of the KDC with which the server
is registered, and (2) a two-component name of type NT-SRV-HST if the
host name is an Internet domain name or a multi-component name of
type NT-SRV-XHST if the name of the host is of a form such as X.500
that allows slash (/) separators. The first component of the two- or
multi-component name will identify the service and the latter
components will identify the host. Where the name of the host is not
case sensitive (for example, with Internet domain names) the name of
the host must be lower case. For services such as telnet and the
Berkeley R commands which run with system privileges, the first
component will be the string "host" instead of a service specific
identifier.

8. Constants and other defined values

8.1. Host address types

All negative values for the host address type are reserved for local
use. All non-negative values are reserved for officially assigned

type fields and interpretations.

The values of the types for the following addresses are chosen to
match the defined address family constants in the Berkeley Standard
Distributions of Unix. They can be found in <sys/socket.h> with
symbolic names AF_xxx (where xxx is an abbreviation of the address
family name).

   Internet addresses

      Internet addresses are 32-bit (4-octet) quantities, encoded in MSB
      order. The type of internet addresses is two (2).

   CHAOSnet addresses

      CHAOSnet addresses are 16-bit (2-octet) quantities, encoded in MSB
      order. The type of CHAOSnet addresses is five (5).

   ISO addresses

      ISO addresses are variable-length. The type of ISO addresses is
      seven (7).

   Xerox Network Services (XNS) addresses

      XNS addresses are 48-bit (6-octet) quantities, encoded in MSB
      order. The type of XNS addresses is six (6).

   AppleTalk Datagram Delivery Protocol (DDP) addresses

      AppleTalk DDP addresses consist of an 8-bit node number and a 16-
      bit network number. The first octet of the address is the node
      number; the remaining two octets encode the network number in MSB
      order. The type of AppleTalk DDP addresses is sixteen (16).

   DECnet Phase IV addresses

      DECnet Phase IV addresses are 16-bit addresses, encoded in LSB
      order. The type of DECnet Phase IV addresses is twelve (12).

8.2. KDC messages

8.2.1. IP transport

   When contacting a Kerberos server (KDC) for a KRB_KDC_REQ request
   using IP transport, the client shall send a UDP datagram containing
   only an encoding of the request to port 88 (decimal) at the KDC's IP

address; the KDC will respond with a reply datagram containing only
an encoding of the reply message (either a KRB_ERROR or a
KRB_KDC_REP) to the sending port at the sender's IP address.

8.2.2. OSI transport

During authentication of an OSI client to and OSI server, the mutual
authentication of an OSI server to an OSI client, the transfer of
credentials from an OSI client to an OSI server, or during exchange
of private or integrity checked messages, Kerberos protocol messages
may be treated as opaque objects and the type of the authentication
mechanism will be:

OBJECT IDENTIFIER ::= {iso (1), org(3), dod(5),internet(1),
                      security(5), kerberosv5(2)}

Depending on the situation, the opaque object will be an
authentication header (KRB_AP_REQ), an authentication reply
(KRB_AP_REP), a safe message (KRB_SAFE), a private message
(KRB_PRIV), or a credentials message (KRB_CRED). The opaque data
contains an application code as specified in the ASN.1 description
for each message. The application code may be used by Kerberos to
determine the message type.

8.2.3. Name of the TGS

The principal identifier of the ticket-granting service shall be
composed of three parts: (1) the realm of the KDC issuing the TGS
ticket (2) a two-part name of type NT-SRVINST, with the first part
"krbtgt" and the second part the name of the realm which will accept
the ticket-granting ticket. For example, a ticket-granting ticket
issued by the ATHENA.MIT.EDU realm to be used to get tickets from the
ATHENA.MIT.EDU KDC has a principal identifier of "ATHENA.MIT.EDU"
(realm), ("krbtgt", "ATHENA.MIT.EDU") (name). A ticket-granting
ticket issued by the ATHENA.MIT.EDU realm to be used to get tickets
from the MIT.EDU realm has a principal identifier of "ATHENA.MIT.EDU"
(realm), ("krbtgt", "MIT.EDU") (name).

8.3. Protocol constants and associated values

The following tables list constants used in the protocol and defines
their meanings.

Encryption type	etype value	block size	minimum pad size	confounder size
NULL	0	1	0	0
des-cbc-crc	1	8	4	8
des-cbc-md4	2	8	0	8
des-cbc-md5	3	8	0	8

Checksum type	sumtype value	checksum size
CRC32	1	4
rsa-md4	2	16
rsa-md4-des	3	24
des-mac	4	16
des-mac-k	5	8
rsa-md4-des-k	6	16
rsa-md5	7	16
rsa-md5-des	8	24

padata type	padata-type value
PA-TGS-REQ	1
PA-ENC-TIMESTAMP	2
PA-PW-SALT	3

authorization data type	ad-type value
reserved values	0-63
OSF-DCE	64
SESAME	65

alternate authentication type	method-type value
reserved values	0-63
ATT-CHALLENGE-RESPONSE	64

transited encoding type	tr-type value
DOMAIN-X500-COMPRESS	1
reserved values	all others

```
--------------+-------+----------------------------------------
Label         |Value  |Meaning or MIT code
--------------+-------+----------------------------------------
```

pvno   5   current Kerberos protocol version number

message types

KRB_AS_REQ   10   Request for initial authentication
KRB_AS_REP   11   Response to KRB_AS_REQ request
KRB_TGS_REQ   12   Request for authentication based on TGT
KRB_TGS_REP   13   Response to KRB_TGS_REQ request
KRB_AP_REQ   14   application request to server
KRB_AP_REP   15   Response to KRB_AP_REQ_MUTUAL
KRB_SAFE   20   Safe (checksummed) application message
KRB_PRIV   21   Private (encrypted) application message
KRB_CRED   22   Private (encrypted) message to forward
                          credentials
KRB_ERROR   30   Error response

name types

KRB_NT_UNKNOWN   0   Name type not known
KRB_NT_PRINCIPAL 1   Just the name of the principal as in DCE, or
                      for users
KRB_NT_SRV_INST   2   Service and other unique instance (krbtgt)
KRB_NT_SRV_HST   3   Service with host name as instance (telnet,
                      rcommands)
KRB_NT_SRV_XHST   4   Service with host as remaining components
KRB_NT_UID   5   Unique ID

error codes

KDC_ERR_NONE   0   No error
KDC_ERR_NAME_EXP   1   Client's entry in database has
                                   expired
KDC_ERR_SERVICE_EXP   2   Server's entry in database has
                                   expired
KDC_ERR_BAD_PVNO   3   Requested protocol version number
                                   not supported
KDC_ERR_C_OLD_MAST_KVNO   4   Client's key encrypted in old
                                   master key
KDC_ERR_S_OLD_MAST_KVNO   5   Server's key encrypted in old
                                   master key
KDC_ERR_C_PRINCIPAL_UNKNOWN   6   Client not found in Kerberos database
KDC_ERR_S_PRINCIPAL_UNKNOWN   7   Server not found in Kerberos database
KDC_ERR_PRINCIPAL_NOT_UNIQUE 8   Multiple principal entries in
                                   database
```

KDC_ERR_NULL_KEY	9	The client or server has a null key
KDC_ERR_CANNOT_POSTDATE	10	Ticket not eligible for postdating
KDC_ERR_NEVER_VALID	11	Requested start time is later than end time
KDC_ERR_POLICY	12	KDC policy rejects request
KDC_ERR_BADOPTION	13	KDC cannot accommodate requested option
KDC_ERR_ETYPE_NOSUPP	14	KDC has no support for encryption type
KDC_ERR_SUMTYPE_NOSUPP	15	KDC has no support for checksum type
KDC_ERR_PADATA_TYPE_NOSUPP	16	KDC has no support for padata type
KDC_ERR_TRTYPE_NOSUPP	17	KDC has no support for transited type
KDC_ERR_CLIENT_REVOKED	18	Clients credentials have been revoked
KDC_ERR_SERVICE_REVOKED	19	Credentials for server have been revoked
KDC_ERR_TGT_REVOKED	20	TGT has been revoked
KDC_ERR_CLIENT_NOTYET	21	Client not yet valid - try again later
KDC_ERR_SERVICE_NOTYET	22	Server not yet valid - try again later
KDC_ERR_KEY_EXPIRED	23	Password has expired - change password to reset
KDC_ERR_PREAUTH_FAILED	24	Pre-authentication information was invalid
KDC_ERR_PREAUTH_REQUIRED	25	Additional pre-authentication required*
KRB_AP_ERR_BAD_INTEGRITY	31	Integrity check on decrypted field failed
KRB_AP_ERR_TKT_EXPIRED	32	Ticket expired
KRB_AP_ERR_TKT_NYV	33	Ticket not yet valid
KRB_AP_ERR_REPEAT	34	Request is a replay
KRB_AP_ERR_NOT_US	35	The ticket isn't for us
KRB_AP_ERR_BADMATCH	36	Ticket and authenticator don't match
KRB_AP_ERR_SKEW	37	Clock skew too great
KRB_AP_ERR_BADADDR	38	Incorrect net address
KRB_AP_ERR_BADVERSION	39	Protocol version mismatch
KRB_AP_ERR_MSG_TYPE	40	Invalid msg type
KRB_AP_ERR_MODIFIED	41	Message stream modified
KRB_AP_ERR_BADORDER	42	Message out of order
KRB_AP_ERR_BADKEYVER	44	Specified version of key is not available
KRB_AP_ERR_NOKEY	45	Service key not available
KRB_AP_ERR_MUT_FAIL	46	Mutual authentication failed
KRB_AP_ERR_BADDIRECTION	47	Incorrect message direction
KRB_AP_ERR_METHOD	48	Alternative authentication method required*
KRB_AP_ERR_BADSEQ	49	Incorrect sequence number in message
KRB_AP_ERR_INAPP_CKSUM	50	Inappropriate type of checksum in

```
                                        message
KRB_ERR_GENERIC                  60     Generic error (description in e-text)
KRB_ERR_FIELD_TOOLONG            61     Field is too long for this
                                        implementation
```

   *This error carries additional information in the e-data field. The
   contents of the e-data field for this message is described in section
   5.9.1.

9. Interoperability requirements

   Version 5 of the Kerberos protocol supports a myriad of options.
   Among these are multiple encryption and checksum types, alternative
   encoding schemes for the transited field, optional mechanisms for
   pre-authentication, the handling of tickets with no addresses,
   options for mutual authentication, user to user authentication,
   support for proxies, forwarding, postdating, and renewing tickets,
   the format of realm names, and the handling of authorization data.

   In order to ensure the interoperability of realms, it is necessary to
   define a minimal configuration which must be supported by all
   implementations. This minimal configuration is subject to change as
   technology does. For example, if at some later date it is discovered
   that one of the required encryption or checksum algorithms is not
   secure, it will be replaced.

9.1. Specification 1

   This section defines the first specification of these options.
   Implementations which are configured in this way can be said to
   support Kerberos Version 5 Specification 1 (5.1).

   Encryption and checksum methods

   The following encryption and checksum mechanisms must be supported.
   Implementations may support other mechanisms as well, but the
   additional mechanisms may only be used when communicating with
   principals known to also support them: Encryption: DES-CBC-MD5
   Checksums: CRC-32, DES-MAC, DES-MAC-K, and DES-MD5

   Realm Names

   All implementations must understand hierarchical realms in both the
   Internet Domain and the X.500 style. When a ticket granting ticket
   for an unknown realm is requested, the KDC must be able to determine
   the names of the intermediate realms between the KDCs realm and the
   requested realm.

Transited field encoding

DOMAIN-X500-COMPRESS (described in section 3.3.3.1) must be
supported. Alternative encodings may be supported, but they may be
used only when that encoding is supported by ALL intermediate realms.

Pre-authentication methods

The TGS-REQ method must be supported. The TGS-REQ method is not used
on the initial request. The PA-ENC-TIMESTAMP method must be supported
by clients but whether it is enabled by default may be determined on
a realm by realm basis. If not used in the initial request and the
error KDC_ERR_PREAUTH_REQUIRED is returned specifying PA-ENCTIMESTAMP
as an acceptable method, the client should retry the initial request
using the PA-ENC-TIMESTAMP preauthentication method. Servers need not
support the PAENC-TIMESTAMP method, but if not supported the server
should ignore the presence of PA-ENC-TIMESTAMP pre-authentication in
a request.

Mutual authentication

Mutual authentication (via the KRB_AP_REP message) must be supported.

Ticket addresses and flags

All KDC's must pass on tickets that carry no addresses (i.e., if a
TGT contains no addresses, the KDC will return derivative tickets),
but each realm may set its own policy for issuing such tickets, and
each application server will set its own policy with respect to
accepting them. By default, servers should not accept them.

Proxies and forwarded tickets must be supported. Individual realms
and application servers can set their own policy on when such tickets
will be accepted.

All implementations must recognize renewable and postdated tickets,
but need not actually implement them. If these options are not
supported, the starttime and endtime in the ticket shall specify a
ticket's entire useful life. When a postdated ticket is decoded by a
server, all implementations shall make the presence of the postdated
flag visible to the calling server.

User-to-user authentication

Support for user to user authentication (via the ENC-TKTIN-SKEY KDC
option) must be provided by implementations, but individual realms
may decide as a matter of policy to reject such requests on a per-
principal or realm-wide basis.

Authorization data

Implementations must pass all authorization data subfields from
ticket-granting tickets to any derivative tickets unless directed to
suppress a subfield as part of the definition of that registered
subfield type (it is never incorrect to pass on a subfield, and no
registered subfield types presently specify suppression at the KDC).

Implementations must make the contents of any authorization data
subfields available to the server when a ticket is used.
Implementations are not required to allow clients to specify the
contents of the authorization data fields.

9.2. Recommended KDC values

Following is a list of recommended values for a KDC implementation,
based on the list of suggested configuration constants (see section
4.4).

minimum lifetime	5 minutes
maximum renewable lifetime	1 week
maximum ticket lifetime	1 day
empty addresses	only when suitable restrictions appear in authorization data
proxiable, etc.	Allowed.

10. Acknowledgments

Early versions of this document, describing version 4 of the
protocol, were written by Jennifer Steiner (formerly at Project
Athena); these drafts provided an excellent starting point for this
current version 5 specification. Many people in the Internet
community have contributed ideas and suggested protocol changes for
version 5. Notable contributions came from Ted Anderson, Steve
Bellovin and Michael Merritt [17], Daniel Bernstein, Mike Burrows,
Donald Davis, Ravi Ganesan, Morrie Gasser, Virgil Gligor, Bill
Griffeth, Mark Lillibridge, Mark Lomas, Steve Lunt, Piers McMahon,
Joe Pato, William Sommerfeld, Stuart Stubblebine, Ralph Swick, Ted
T'so, and Stanley Zanarotti. Many others commented and helped shape
this specification into its current form.

11. References

[1] Miller, S., Neuman, C., Schiller, J., and J. Saltzer, "Section
     E.2.1: Kerberos Authentication and Authorization System",
     M.I.T. Project Athena, Cambridge, Massachusetts, December 21,
     1987.

[2] Steiner, J., Neuman, C., and J. Schiller, "Kerberos: An
     Authentication Service for Open Network Systems", pp. 191-202 in
     Usenix Conference Proceedings, Dallas, Texas, February, 1988.

[3] Needham, R., and M. Schroeder, "Using Encryption for
     Authentication in Large Networks of Computers", Communications
     of the ACM, Vol. 21 (12), pp. 993-999, December 1978.

[4] Denning, D., and G. Sacco, "Time stamps in Key Distribution
     Protocols", Communications of the ACM, Vol. 24 (8), pp. 533-536,
     August 1981.

[5] Kohl, J., Neuman, C., and T. Ts'o, "The Evolution of the
     Kerberos Authentication Service", in an IEEE Computer Society
     Text soon to be published, June 1992.

[6] Davis, D., and R. Swick, "Workstation Services and Kerberos
     Authentication at Project Athena", Technical Memorandum TM-424,
     MIT Laboratory for Computer Science, February 1990.

[7] Levine, P., Gretzinger, M, Diaz, J., Sommerfeld, W., and K.
     Raeburn, "Section E.1: Service Management System, M.I.T.
     Project Athena, Cambridge, Mas sachusetts (1987).

[8] CCITT, Recommendation X.509: The Directory Authentication
     Framework, December 1988.

[9] Neuman, C., "Proxy-Based Authorization and Accounting for
     Distributed Systems," in Proceedings of the 13th International
     Conference on Distributed Computing Systems", Pittsburgh, PA,
     May 1993.

[10] Pato, J., "Using Pre-Authentication to Avoid Password Guessing
     Attacks", Open Software Foundation DCE Request for Comments 26,
     December 1992.

[11] National Bureau of Standards, U.S. Department of Commerce, "Data
     Encryption Standard", Federal Information Processing Standards
     Publication 46, Washington, DC (1977).

[12] National Bureau of Standards, U.S. Department of Commerce, "DES
     Modes of Operation", Federal Information Processing Standards
     Publication 81, Springfield, VA, December 1980.

[13] Stubblebine S., and V. Gligor, "On Message Integrity in
     Cryptographic Protocols", in Proceedings of the IEEE Symposium
     on Research in Security and Privacy, Oakland, California, May
     1992.

[14] International Organization for Standardization, "ISO Information
     Processing Systems - Data Communication High-Level Data Link
     Control Procedure - Frame Structure", IS 3309, October 1984, 3rd
     Edition.

[15] Rivest, R., "The MD4 Message Digest Algorithm", RFC 1320, MIT
     Laboratory for Computer Science, April 1992.

[16] Rivest, R., "The MD5 Message Digest Algorithm", RFC 1321, MIT
     Laboratory for Computer Science, April 1992.

[17] Bellovin S., and M. Merritt, "Limitations of the Kerberos
     Authentication System", Computer Communications Review, Vol.
     20(5), pp. 119-132, October 1990.

12. Security Considerations

   Security issues are discussed throughout this memo.

13. Authors' Addresses

   John Kohl
   Digital Equipment Corporation
   110 Spit Brook Road, M/S ZKO3-3/U14
   Nashua, NH   03062

   Phone: 603-881-2481
   EMail: jtkohl@zk3.dec.com

   B. Clifford Neuman
   USC/Information Sciences Institute
   4676 Admiralty Way #1001
   Marina del Rey, CA 90292-6695

   Phone: 310-822-1511
   EMail: bcn@isi.edu

A. Pseudo-code for protocol processing

   This appendix provides pseudo-code describing how the messages are to
   be constructed and interpreted by clients and servers.

A.1. KRB_AS_REQ generation

```
    request.pvno := protocol version; /* pvno = 5 */
    request.msg-type := message type; /* type = KRB_AS_REQ */

    if(pa_enc_timestamp_required) then
            request.padata.padata-type = PA-ENC-TIMESTAMP;
            get system_time;
            padata-body.patimestamp,pausec = system_time;
            encrypt padata-body into request.padata.padata-value
                    using client.key; /* derived from password */
    endif

    body.kdc-options := users's preferences;
    body.cname := user's name;
    body.realm := user's realm;
    body.sname := service's name; /* usually "krbtgt",
                                      "localrealm" */
    if (body.kdc-options.POSTDATED is set) then
            body.from := requested starting time;
    else
            omit body.from;
    endif
    body.till := requested end time;
    if (body.kdc-options.RENEWABLE is set) then
            body.rtime := requested final renewal time;
    endif
    body.nonce := random_nonce();
    body.etype := requested etypes;
    if (user supplied addresses) then
            body.addresses := user's addresses;
    else
            omit body.addresses;
    endif
    omit body.enc-authorization-data;
    request.req-body := body;

    kerberos := lookup(name of local kerberos server (or servers));
    send(packet,kerberos);

    wait(for response);
    if (timed_out) then
            retry or use alternate server;
    endif
```

A.2. KRB_AS_REQ verification and KRB_AS_REP generation
          decode message into req;

          client := lookup(req.cname,req.realm);
          server := lookup(req.sname,req.realm);
          get system_time;
          kdc_time := system_time.seconds;

          if (!client) then
                  /* no client in Database */
                  error_out(KDC_ERR_C_PRINCIPAL_UNKNOWN);
          endif
          if (!server) then
                  /* no server in Database */
                  error_out(KDC_ERR_S_PRINCIPAL_UNKNOWN);
          endif

          if(client.pa_enc_timestamp_required and
             pa_enc_timestamp not present) then
                  error_out(KDC_ERR_PREAUTH_REQUIRED(PA_ENC_TIMESTAMP));
          endif

          if(pa_enc_timestamp present) then
                  decrypt req.padata-value into decrypted_enc_timestamp
                          using client.key;
                          using auth_hdr.authenticator.subkey;
                  if (decrypt_error()) then
                          error_out(KRB_AP_ERR_BAD_INTEGRITY);
                  if(decrypted_enc_timestamp is not within allowable
                          skew) then error_out(KDC_ERR_PREAUTH_FAILED);
                  endif
                  if(decrypted_enc_timestamp and usec is replay)
                          error_out(KDC_ERR_PREAUTH_FAILED);
                  endif
                  add decrypted_enc_timestamp and usec to replay cache;
          endif

          use_etype := first supported etype in req.etypes;

          if (no support for req.etypes) then
                  error_out(KDC_ERR_ETYPE_NOSUPP);
          endif

          new_tkt.vno := ticket version; /* = 5 */
          new_tkt.sname := req.sname;
          new_tkt.srealm := req.srealm;
          reset all flags in new_tkt.flags;

```
     /* It should be noted that local policy may affect the  */
     /* processing of any of these flags.  For example, some */
     /* realms may refuse to issue renewable tickets         */

     if (req.kdc-options.FORWARDABLE is set) then
             set new_tkt.flags.FORWARDABLE;
     endif
     if (req.kdc-options.PROXIABLE is set) then
             set new_tkt.flags.PROXIABLE;
     endif
     if (req.kdc-options.ALLOW-POSTDATE is set) then
             set new_tkt.flags.ALLOW-POSTDATE;
     endif
     if ((req.kdc-options.RENEW is set) or
         (req.kdc-options.VALIDATE is set) or
         (req.kdc-options.PROXY is set) or
         (req.kdc-options.FORWARDED is set) or
         (req.kdc-options.ENC-TKT-IN-SKEY is set)) then
             error_out(KDC_ERR_BADOPTION);
     endif

     new_tkt.session := random_session_key();
     new_tkt.cname := req.cname;
     new_tkt.crealm := req.crealm;
     new_tkt.transited := empty_transited_field();

     new_tkt.authtime := kdc_time;

     if (req.kdc-options.POSTDATED is set) then
        if (against_postdate_policy(req.from)) then
            error_out(KDC_ERR_POLICY);
        endif
        set new_tkt.flags.INVALID;
        new_tkt.starttime := req.from;
     else
        omit new_tkt.starttime; /* treated as authtime when
                                    omitted */
     endif
     if (req.till = 0) then
             till := infinity;
     else
             till := req.till;
     endif

     new_tkt.endtime := min(till,
                            new_tkt.starttime+client.max_life,
                            new_tkt.starttime+server.max_life,
                            new_tkt.starttime+max_life_for_realm);
```

```
     if ((req.kdc-options.RENEWABLE-OK is set) and
        (new_tkt.endtime < req.till)) then
             /* we set the RENEWABLE option for later processing */
             set req.kdc-options.RENEWABLE;
             req.rtime := req.till;
     endif

     if (req.rtime = 0) then
             rtime := infinity;
     else
             rtime := req.rtime;
     endif

     if (req.kdc-options.RENEWABLE is set) then
             set new_tkt.flags.RENEWABLE;
             new_tkt.renew-till := min(rtime,
             new_tkt.starttime+client.max_rlife,
             new_tkt.starttime+server.max_rlife,
             new_tkt.starttime+max_rlife_for_realm);
     else
             omit new_tkt.renew-till; /* only present if RENEWABLE */
     endif

     if (req.addresses) then
             new_tkt.caddr := req.addresses;
     else
             omit new_tkt.caddr;
     endif

     new_tkt.authorization_data := empty_authorization_data();

     encode to-be-encrypted part of ticket into OCTET STRING;
     new_tkt.enc-part := encrypt OCTET STRING
         using etype_for_key(server.key), server.key, server.p_kvno;


     /* Start processing the response */

     resp.pvno := 5;
     resp.msg-type := KRB_AS_REP;
     resp.cname := req.cname;
     resp.crealm := req.realm;
     resp.ticket := new_tkt;

     resp.key := new_tkt.session;
     resp.last-req := fetch_last_request_info(client);
     resp.nonce := req.nonce;
     resp.key-expiration := client.expiration;
```

```
        resp.flags := new_tkt.flags;

        resp.authtime := new_tkt.authtime;
        resp.starttime := new_tkt.starttime;
        resp.endtime := new_tkt.endtime;

        if (new_tkt.flags.RENEWABLE) then
                resp.renew-till := new_tkt.renew-till;
        endif

        resp.realm := new_tkt.realm;
        resp.sname := new_tkt.sname;

        resp.caddr := new_tkt.caddr;

        encode body of reply into OCTET STRING;

        resp.enc-part := encrypt OCTET STRING
                        using use_etype, client.key, client.p_kvno;
        send(resp);

A.3.  KRB_AS_REP verification
        decode response into resp;

        if (resp.msg-type = KRB_ERROR) then
                if(error = KDC_ERR_PREAUTH_REQUIRED(PA_ENC_TIMESTAMP))
                        then set pa_enc_timestamp_required;
                        goto KRB_AS_REQ;
                endif
                process_error(resp);
                return;
        endif

        /* On error, discard the response, and zero the session key */
        /* from the response immediately */

        key = get_decryption_key(resp.enc-part.kvno, resp.enc-part.etype,
                                resp.padata);
        unencrypted part of resp := decode of decrypt of resp.enc-part
                                using resp.enc-part.etype and key;
        zero(key);

        if (common_as_rep_tgs_rep_checks fail) then
                destroy resp.key;
                return error;
        endif

        if near(resp.princ_exp) then
```

```
                   print(warning message);
           endif
           save_for_later(ticket,session,client,server,times,flags);


A.4.   KRB_AS_REP and KRB_TGS_REP common checks
           if (decryption_error() or
               (req.cname != resp.cname) or
               (req.realm != resp.crealm) or
               (req.sname != resp.sname) or
               (req.realm != resp.realm) or
               (req.nonce != resp.nonce) or
               (req.addresses != resp.caddr)) then
                   destroy resp.key;
                   return KRB_AP_ERR_MODIFIED;
           endif

           /* make sure no flags are set that shouldn't be, and that  */
           /* all that should be are set                              */
           if (!check_flags_for_compatability(req.kdc-options,resp.flags)
                   then destroy resp.key;
                   return KRB_AP_ERR_MODIFIED;
           endif

           if ((req.from = 0) and
               (resp.starttime is not within allowable skew)) then
                   destroy resp.key;
                   return KRB_AP_ERR_SKEW;
           endif
           if ((req.from != 0) and (req.from != resp.starttime)) then
                   destroy resp.key;
                   return KRB_AP_ERR_MODIFIED;
           endif
           if ((req.till != 0) and (resp.endtime > req.till)) then
                   destroy resp.key;
                   return KRB_AP_ERR_MODIFIED;
           endif

           if ((req.kdc-options.RENEWABLE is set) and
               (req.rtime != 0) and (resp.renew-till > req.rtime)) then
                   destroy resp.key;
                   return KRB_AP_ERR_MODIFIED;
           endif
           if ((req.kdc-options.RENEWABLE-OK is set) and
               (resp.flags.RENEWABLE) and
               (req.till != 0) and
               (resp.renew-till > req.till)) then
                   destroy resp.key;
                   return KRB_AP_ERR_MODIFIED;
```

```
        endif

A.5.  KRB_TGS_REQ generation
        /* Note that make_application_request might have to      */
        /* recursivly call this routine to get the appropriate  */
        /* ticket-granting ticket                                */

        request.pvno := protocol version; /* pvno = 5 */
        request.msg-type := message type; /* type = KRB_TGS_REQ */

        body.kdc-options := users's preferences;
        /* If the TGT is not for the realm of the end-server  */
        /* then the sname will be for a TGT for the end-realm */
        /* and the realm of the requested ticket (body.realm) */
        /* will be that of the TGS to which the TGT we are    */
        /* sending applies                                    */
        body.sname := service's name;
        body.realm := service's realm;

        if (body.kdc-options.POSTDATED is set) then
                body.from := requested starting time;
        else
                omit body.from;
        endif
        body.till := requested end time;
        if (body.kdc-options.RENEWABLE is set) then
                body.rtime := requested final renewal time;
        endif
        body.nonce := random_nonce();
        body.etype := requested etypes;
        if (user supplied addresses) then
                body.addresses := user's addresses;
        else
                omit body.addresses;
        endif

        body.enc-authorization-data := user-supplied data;
        if (body.kdc-options.ENC-TKT-IN-SKEY) then
                body.additional-tickets_ticket := second TGT;
        endif

        request.req-body := body;
        check := generate_checksum (req.body,checksumtype);

        request.padata[0].padata-type := PA-TGS-REQ;
        request.padata[0].padata-value := create a KRB_AP_REQ using
                                        the TGT and checksum
```

```
        /* add in any other padata as required/supplied */

        kerberos := lookup(name of local kerberose server (or servers));
        send(packet,kerberos);

        wait(for response);
        if (timed_out) then
                retry or use alternate server;
        endif
```

A.6. KRB_TGS_REQ verification and KRB_TGS_REP generation

```
        /* note that reading the application request requires first
        determining the server for which a ticket was issued, and
        choosing the correct key for decryption.  The name of the
        server appears in the plaintext part of the ticket. */

        if (no KRB_AP_REQ in req.padata) then
                error_out(KDC_ERR_PADATA_TYPE_NOSUPP);
        endif
        verify KRB_AP_REQ in req.padata;

        /* Note that the realm in which the Kerberos server is
        operating is determined by the instance from the
        ticket-granting ticket.  The realm in the ticket-granting
        ticket is the realm under which the ticket granting ticket was
        issued.  It is possible for a single Kerberos server to
        support more than one realm. */

        auth_hdr := KRB_AP_REQ;
        tgt := auth_hdr.ticket;

        if (tgt.sname is not a TGT for local realm and is not
                req.sname) then error_out(KRB_AP_ERR_NOT_US);

        realm := realm_tgt_is_for(tgt);

        decode remainder of request;

        if (auth_hdr.authenticator.cksum is missing) then
                error_out(KRB_AP_ERR_INAPP_CKSUM);
        endif
        if (auth_hdr.authenticator.cksum type is not supported) then
                error_out(KDC_ERR_SUMTYPE_NOSUPP);
        endif
        if (auth_hdr.authenticator.cksum is not both collision-proof
           and keyed)  then
                error_out(KRB_AP_ERR_INAPP_CKSUM);
        endif
```

```
        set computed_checksum := checksum(req);
        if (computed_checksum != auth_hdr.authenticatory.cksum) then
                error_out(KRB_AP_ERR_MODIFIED);
        endif

        server := lookup(req.sname,realm);

        if (!server) then
                if (is_foreign_tgt_name(server)) then
                        server := best_intermediate_tgs(server);
                else
                        /* no server in Database */
                        error_out(KDC_ERR_S_PRINCIPAL_UNKNOWN);
                endif
        endif

        session := generate_random_session_key();


        use_etype := first supported etype in req.etypes;

        if (no support for req.etypes) then
                error_out(KDC_ERR_ETYPE_NOSUPP);
        endif

        new_tkt.vno := ticket version; /* = 5 */
        new_tkt.sname := req.sname;
        new_tkt.srealm := realm;
        reset all flags in new_tkt.flags;

        /* It should be noted that local policy may affect the  */
        /* processing of any of these flags.  For example, some */
        /* realms may refuse to issue renewable tickets         */

        new_tkt.caddr := tgt.caddr;
        resp.caddr := NULL; /* We only include this if they change */
        if (req.kdc-options.FORWARDABLE is set) then
                if (tgt.flags.FORWARDABLE is reset) then
                        error_out(KDC_ERR_BADOPTION);
                endif
                set new_tkt.flags.FORWARDABLE;
        endif
        if (req.kdc-options.FORWARDED is set) then
                if (tgt.flags.FORWARDABLE is reset) then
                        error_out(KDC_ERR_BADOPTION);
                endif
                set new_tkt.flags.FORWARDED;
                new_tkt.caddr := req.addresses;
```

```
                    resp.caddr := req.addresses;
            endif
            if (tgt.flags.FORWARDED is set) then
                    set new_tkt.flags.FORWARDED;
            endif

            if (req.kdc-options.PROXIABLE is set) then
                    if (tgt.flags.PROXIABLE is reset)
                            error_out(KDC_ERR_BADOPTION);
                    endif
                    set new_tkt.flags.PROXIABLE;
            endif
            if (req.kdc-options.PROXY is set) then
                    if (tgt.flags.PROXIABLE is reset) then
                            error_out(KDC_ERR_BADOPTION);
                    endif
                    set new_tkt.flags.PROXY;
                    new_tkt.caddr := req.addresses;
                    resp.caddr := req.addresses;
            endif

            if (req.kdc-options.POSTDATE is set) then
                    if (tgt.flags.POSTDATE is reset)
                            error_out(KDC_ERR_BADOPTION);
                    endif
                    set new_tkt.flags.POSTDATE;
            endif
            if (req.kdc-options.POSTDATED is set) then
                    if (tgt.flags.POSTDATE is reset) then
                            error_out(KDC_ERR_BADOPTION);
                    endif
                    set new_tkt.flags.POSTDATED;
                    set new_tkt.flags.INVALID;
                    if (against_postdate_policy(req.from)) then
                            error_out(KDC_ERR_POLICY);
                    endif
                    new_tkt.starttime := req.from;
            endif


            if (req.kdc-options.VALIDATE is set) then
                    if (tgt.flags.INVALID is reset) then
                            error_out(KDC_ERR_POLICY);
                    endif
                    if (tgt.starttime > kdc_time) then
                            error_out(KRB_AP_ERR_NYV);
                    endif
                    if (check_hot_list(tgt)) then
```

```
                            error_out(KRB_AP_ERR_REPEAT);
                 endif
                 tkt := tgt;
                 reset new_tkt.flags.INVALID;
         endif

         if (req.kdc-options.(any flag except ENC-TKT-IN-SKEY, RENEW,
                            and those already processed) is set) then
                 error_out(KDC_ERR_BADOPTION);
         endif

         new_tkt.authtime := tgt.authtime;

         if (req.kdc-options.RENEW is set) then
           /* Note that if the endtime has already passed, the ticket */
           /* would have been rejected in the initial authentication  */
           /* stage, so there is no need to check again here          */
                 if (tgt.flags.RENEWABLE is reset) then
                         error_out(KDC_ERR_BADOPTION);
                 endif
                 if (tgt.renew-till >= kdc_time) then
                         error_out(KRB_AP_ERR_TKT_EXPIRED);
                 endif
                 tkt := tgt;
                 new_tkt.starttime := kdc_time;
                 old_life := tgt.endtime - tgt.starttime;
                 new_tkt.endtime := min(tgt.renew-till,
                                        new_tkt.starttime + old_life);
         else
                 new_tkt.starttime := kdc_time;
                 if (req.till = 0) then
                         till := infinity;
                 else
                         till := req.till;
                 endif
                 new_tkt.endtime := min(till,
                                    new_tkt.starttime+client.max_life,
                                    new_tkt.starttime+server.max_life,
                                    new_tkt.starttime+max_life_for_realm,
                                    tgt.endtime);

                 if ((req.kdc-options.RENEWABLE-OK is set) and
                     (new_tkt.endtime < req.till) and
                     (tgt.flags.RENEWABLE is set) then
                         /* we set the RENEWABLE option for later  */
                         /* processing                             */
                         set req.kdc-options.RENEWABLE;
                         req.rtime := min(req.till, tgt.renew-till);
```

```
                endif
        endif

        if (req.rtime = 0) then
                rtime := infinity;
        else
                rtime := req.rtime;
        endif

        if ((req.kdc-options.RENEWABLE is set) and
            (tgt.flags.RENEWABLE is set)) then
                set new_tkt.flags.RENEWABLE;
                new_tkt.renew-till := min(rtime,
                new_tkt.starttime+client.max_rlife,
                new_tkt.starttime+server.max_rlife,
                new_tkt.starttime+max_rlife_for_realm,
                tgt.renew-till);
        else
                new_tkt.renew-till := OMIT;
                                /* leave the renew-till field out */
        endif
        if (req.enc-authorization-data is present) then
                decrypt req.enc-authorization-data
                        into    decrypted_authorization_data
                        using auth_hdr.authenticator.subkey;
                if (decrypt_error()) then
                        error_out(KRB_AP_ERR_BAD_INTEGRITY);
                endif
        endif
        new_tkt.authorization_data :=
        req.auth_hdr.ticket.authorization_data +
                                decrypted_authorization_data;

        new_tkt.key := session;
        new_tkt.crealm := tgt.crealm;
        new_tkt.cname := req.auth_hdr.ticket.cname;

        if (realm_tgt_is_for(tgt) := tgt.realm) then
                /* tgt issued by local realm */
                new_tkt.transited := tgt.transited;
        else
                /* was issued for this realm by some other realm */
                if (tgt.transited.tr-type not supported) then
                        error_out(KDC_ERR_TRTYPE_NOSUPP);
                endif
                new_tkt.transited
                   := compress_transited(tgt.transited + tgt.realm)
        endif
```

```
        encode encrypted part of new_tkt into OCTET STRING;
        if (req.kdc-options.ENC-TKT-IN-SKEY is set) then
                if (server not specified) then
                        server = req.second_ticket.client;
                endif
                if ((req.second_ticket is not a TGT) or
                    (req.second_ticket.client != server)) then
                        error_out(KDC_ERR_POLICY);
                endif

                new_tkt.enc-part := encrypt OCTET STRING using
                        using etype_for_key(second-ticket.key),
                                                second-ticket.key;
        else
                new_tkt.enc-part := encrypt OCTET STRING
                        using etype_for_key(server.key), server.key,
                                                server.p_kvno;
        endif

        resp.pvno := 5;
        resp.msg-type := KRB_TGS_REP;
        resp.crealm := tgt.crealm;
        resp.cname := tgt.cname;
        resp.ticket := new_tkt;

        resp.key := session;
        resp.nonce := req.nonce;
        resp.last-req := fetch_last_request_info(client);
        resp.flags := new_tkt.flags;

        resp.authtime := new_tkt.authtime;
        resp.starttime := new_tkt.starttime;
        resp.endtime := new_tkt.endtime;

        omit resp.key-expiration;

        resp.sname := new_tkt.sname;
        resp.realm := new_tkt.realm;

        if (new_tkt.flags.RENEWABLE) then
                resp.renew-till := new_tkt.renew-till;
        endif


        encode body of reply into OCTET STRING;

        if (req.padata.authenticator.subkey)
                resp.enc-part := encrypt OCTET STRING using use_etype,
```

```
                                req.padata.authenticator.subkey;
        else resp.enc-part := encrypt OCTET STRING
                                using use_etype, tgt.key;

        send(resp);
```

A.7. KRB_TGS_REP verification
```
        decode response into resp;

        if (resp.msg-type = KRB_ERROR) then
                process_error(resp);
                return;
        endif

        /* On error, discard the response, and zero the session key from
        the response immediately */

        if (req.padata.authenticator.subkey)
                unencrypted part of resp :=
                        decode of decrypt of resp.enc-part
                        using resp.enc-part.etype and subkey;
        else unencrypted part of resp :=
                        decode of decrypt of resp.enc-part
                        using resp.enc-part.etype and tgt's session key;
        if (common_as_rep_tgs_rep_checks fail) then
                destroy resp.key;
                return error;
        endif

        check authorization_data as necessary;
        save_for_later(ticket,session,client,server,times,flags);
```

A.8. Authenticator generation
```
        body.authenticator-vno := authenticator vno; /* = 5 */
        body.cname, body.crealm := client name;
        if (supplying checksum) then
                body.cksum := checksum;
        endif
        get system_time;
        body.ctime, body.cusec := system_time;
        if (selecting sub-session key) then
                select sub-session key;
                body.subkey := sub-session key;
        endif
        if (using sequence numbers) then
                select initial sequence number;
                body.seq-number := initial sequence;
        endif
```

A.9.   KRB_AP_REQ generation
        obtain ticket and session_key from cache;

        packet.pvno := protocol version; /* 5 */
        packet.msg-type := message type; /* KRB_AP_REQ */

        if (desired(MUTUAL_AUTHENTICATION)) then
                set packet.ap-options.MUTUAL-REQUIRED;
        else
                reset packet.ap-options.MUTUAL-REQUIRED;
        endif
        if (using session key for ticket) then
                set packet.ap-options.USE-SESSION-KEY;
        else
                reset packet.ap-options.USE-SESSION-KEY;
        endif
        packet.ticket := ticket; /* ticket */
        generate authenticator;
        encode authenticator into OCTET STRING;
        encrypt OCTET STRING into packet.authenticator
                            using session_key;

A.10.   KRB_AP_REQ verification
        receive packet;
        if (packet.pvno != 5) then
                either process using other protocol spec
                or error_out(KRB_AP_ERR_BADVERSION);
        endif
        if (packet.msg-type != KRB_AP_REQ) then
                error_out(KRB_AP_ERR_MSG_TYPE);
        endif
        if (packet.ticket.tkt_vno != 5) then
                either process using other protocol spec
                or error_out(KRB_AP_ERR_BADVERSION);
        endif
        if (packet.ap_options.USE-SESSION-KEY is set) then
                retrieve session key from ticket-granting ticket for
                 packet.ticket.{sname,srealm,enc-part.etype};
        else
           retrieve service key for
           packet.ticket.{sname,srealm,enc-part.etype,enc-part.skvno};
        endif
        if (no_key_available) then
                if (cannot_find_specified_skvno) then
                        error_out(KRB_AP_ERR_BADKEYVER);
                else
                        error_out(KRB_AP_ERR_NOKEY);
                endif

```
        endif
        decrypt packet.ticket.enc-part into decr_ticket
                                        using retrieved key;
        if (decryption_error()) then
                error_out(KRB_AP_ERR_BAD_INTEGRITY);
        endif
        decrypt packet.authenticator into decr_authenticator
                using decr_ticket.key;
        if (decryption_error()) then
                error_out(KRB_AP_ERR_BAD_INTEGRITY);
        endif
        if (decr_authenticator.{cname,crealm} !=
            decr_ticket.{cname,crealm}) then
                error_out(KRB_AP_ERR_BADMATCH);
        endif
        if (decr_ticket.caddr is present) then
                if (sender_address(packet) is not in decr_ticket.caddr)
                        then error_out(KRB_AP_ERR_BADADDR);
                endif
        elseif (application requires addresses) then
                error_out(KRB_AP_ERR_BADADDR);
        endif
        if (not in_clock_skew(decr_authenticator.ctime,
                            decr_authenticator.cusec)) then
                error_out(KRB_AP_ERR_SKEW);
        endif
        if (repeated(decr_authenticator.{ctime,cusec,cname,crealm}))
                then error_out(KRB_AP_ERR_REPEAT);
        endif
        save_identifier(decr_authenticator.{ctime,cusec,cname,crealm});
        get system_time;
        if ((decr_ticket.starttime-system_time > CLOCK_SKEW) or
            (decr_ticket.flags.INVALID is set)) then
                /* it hasn't yet become valid */
                error_out(KRB_AP_ERR_TKT_NYV);
        endif
        if (system_time-decr_ticket.endtime > CLOCK_SKEW) then
                error_out(KRB_AP_ERR_TKT_EXPIRED);
        endif
        /* caller must check decr_ticket.flags for any pertinent */
        /* details */
        return(OK, decr_ticket, packet.ap_options.MUTUAL-REQUIRED);

A.11.   KRB_AP_REP generation
        packet.pvno := protocol version; /* 5 */
        packet.msg-type := message type; /* KRB_AP_REP */
        body.ctime := packet.ctime;
        body.cusec := packet.cusec;
```

```
        if (selecting sub-session key) then
                select sub-session key;
                body.subkey := sub-session key;
        endif
        if (using sequence numbers) then
                select initial sequence number;
                body.seq-number := initial sequence;
        endif

        encode body into OCTET STRING;

        select encryption type;
        encrypt OCTET STRING into packet.enc-part;
```

A.12.   KRB_AP_REP verification
```
        receive packet;
        if (packet.pvno != 5) then
                either process using other protocol spec
                or error_out(KRB_AP_ERR_BADVERSION);
        endif
        if (packet.msg-type != KRB_AP_REP) then
                error_out(KRB_AP_ERR_MSG_TYPE);
        endif
        cleartext := decrypt(packet.enc-part)
                     using ticket's session key;
        if (decryption_error()) then
                error_out(KRB_AP_ERR_BAD_INTEGRITY);
        endif
        if (cleartext.ctime != authenticator.ctime) then
                error_out(KRB_AP_ERR_MUT_FAIL);
        endif
        if (cleartext.cusec != authenticator.cusec) then
                error_out(KRB_AP_ERR_MUT_FAIL);
        endif
        if (cleartext.subkey is present) then
                save cleartext.subkey for future use;
        endif
        if (cleartext.seq-number is present) then
                save cleartext.seq-number for future verifications;
        endif
        return(AUTHENTICATION_SUCCEEDED);
```

A.13.   KRB_SAFE generation
```
        collect user data in buffer;

        /* assemble packet: */
        packet.pvno := protocol version; /* 5 */
        packet.msg-type := message type; /* KRB_SAFE */
```

```
        body.user-data := buffer; /* DATA */
        if (using timestamp) then
                get system_time;
                body.timestamp, body.usec := system_time;
        endif
        if (using sequence numbers) then
                body.seq-number := sequence number;
        endif
        body.s-address := sender host addresses;
        if (only one recipient) then
                body.r-address := recipient host address;
        endif
        checksum.cksumtype := checksum type;
        compute checksum over body;
        checksum.checksum := checksum value; /* checksum.checksum */
        packet.cksum := checksum;
        packet.safe-body := body;

A.14.   KRB_SAFE verification
        receive packet;
        if (packet.pvno != 5) then
                either process using other protocol spec
                or error_out(KRB_AP_ERR_BADVERSION);
        endif
        if (packet.msg-type != KRB_SAFE) then
                error_out(KRB_AP_ERR_MSG_TYPE);
        endif
        if (packet.checksum.cksumtype is not both collision-proof
                                            and keyed) then
                error_out(KRB_AP_ERR_INAPP_CKSUM);
        endif
        if (safe_priv_common_checks_ok(packet)) then
                set computed_checksum := checksum(packet.body);
                if (computed_checksum != packet.checksum) then
                        error_out(KRB_AP_ERR_MODIFIED);
                endif
                return (packet, PACKET_IS_GENUINE);
        else
                return common_checks_error;
        endif

A.15.   KRB_SAFE and KRB_PRIV common checks
        if (packet.s-address != O/S_sender(packet)) then
            /* O/S report of sender not who claims to have sent it */
            error_out(KRB_AP_ERR_BADADDR);
        endif
        if ((packet.r-address is present) and
            (packet.r-address != local_host_address)) then
```

```
                    /* was not sent to proper place */
                    error_out(KRB_AP_ERR_BADADDR);
          endif
          if (((packet.timestamp is present) and
              (not in_clock_skew(packet.timestamp,packet.usec))) or
             (packet.timestamp is not present and timestamp expected))
                    then error_out(KRB_AP_ERR_SKEW);
          endif
          if (repeated(packet.timestamp,packet.usec,packet.s-address))
                    then error_out(KRB_AP_ERR_REPEAT);
          endif
          if (((packet.seq-number is present) and
              ((not in_sequence(packet.seq-number)))) or
             (packet.seq-number is not present and sequence expected))
                    then error_out(KRB_AP_ERR_BADORDER);
          endif
          if (packet.timestamp not present and
              packet.seq-number not present) then
                    error_out(KRB_AP_ERR_MODIFIED);
          endif

          save_identifier(packet.{timestamp,usec,s-address},
                          sender_principal(packet));

          return PACKET_IS_OK;

A.16.  KRB_PRIV generation
       collect user data in buffer;

       /* assemble packet: */
       packet.pvno := protocol version; /* 5 */
       packet.msg-type := message type; /* KRB_PRIV */

       packet.enc-part.etype := encryption type;

       body.user-data := buffer;
       if (using timestamp) then
               get system_time;
               body.timestamp, body.usec := system_time;
       endif
       if (using sequence numbers) then
               body.seq-number := sequence number;
       endif
       body.s-address := sender host addresses;
       if (only one recipient) then
               body.r-address := recipient host address;
       endif
```

```
        encode body into OCTET STRING;

        select encryption type;
        encrypt OCTET STRING into packet.enc-part.cipher;
```

A.17.   KRB_PRIV verification
```
        receive packet;
        if (packet.pvno != 5) then
                either process using other protocol spec
                or error_out(KRB_AP_ERR_BADVERSION);
        endif
        if (packet.msg-type != KRB_PRIV) then
                error_out(KRB_AP_ERR_MSG_TYPE);
        endif

        cleartext := decrypt(packet.enc-part) using negotiated key;
        if (decryption_error()) then
                error_out(KRB_AP_ERR_BAD_INTEGRITY);
        endif

        if (safe_priv_common_checks_ok(cleartext)) then
            return(cleartext.DATA, PACKET_IS_GENUINE_AND_UNMODIFIED);
        else
                return common_checks_error;
        endif
```

A.18.   KRB_CRED generation
```
        invoke KRB_TGS; /* obtain tickets to be provided to peer */

        /* assemble packet: */
        packet.pvno := protocol version; /* 5 */
        packet.msg-type := message type; /* KRB_CRED */

        for (tickets[n] in tickets to be forwarded) do
                packet.tickets[n] = tickets[n].ticket;
        done

        packet.enc-part.etype := encryption type;

        for (ticket[n] in tickets to be forwarded) do
                body.ticket-info[n].key = tickets[n].session;
                body.ticket-info[n].prealm = tickets[n].crealm;
                body.ticket-info[n].pname = tickets[n].cname;
                body.ticket-info[n].flags = tickets[n].flags;
                body.ticket-info[n].authtime = tickets[n].authtime;
                body.ticket-info[n].starttime = tickets[n].starttime;
                body.ticket-info[n].endtime = tickets[n].endtime;
                body.ticket-info[n].renew-till = tickets[n].renew-till;
```

```
                body.ticket-info[n].srealm = tickets[n].srealm;
                body.ticket-info[n].sname = tickets[n].sname;
                body.ticket-info[n].caddr = tickets[n].caddr;
        done

        get system_time;
        body.timestamp, body.usec := system_time;

        if (using nonce) then
                body.nonce := nonce;
        endif

        if (using s-address) then
                body.s-address := sender host addresses;
        endif
        if (limited recipients) then
                body.r-address := recipient host address;
        endif

        encode body into OCTET STRING;

        select encryption type;
        encrypt OCTET STRING into packet.enc-part.cipher
        using negotiated encryption key;

A.19.   KRB_CRED verification
        receive packet;
        if (packet.pvno != 5) then
                either process using other protocol spec
                or error_out(KRB_AP_ERR_BADVERSION);
        endif
        if (packet.msg-type != KRB_CRED) then
                error_out(KRB_AP_ERR_MSG_TYPE);
        endif

        cleartext := decrypt(packet.enc-part) using negotiated key;
        if (decryption_error()) then
                error_out(KRB_AP_ERR_BAD_INTEGRITY);
        endif
        if ((packet.r-address is present or required) and
           (packet.s-address != O/S_sender(packet)) then
            /* O/S report of sender not who claims to have sent it */
            error_out(KRB_AP_ERR_BADADDR);
        endif
        if ((packet.r-address is present) and
           (packet.r-address != local_host_address)) then
                /* was not sent to proper place */
                error_out(KRB_AP_ERR_BADADDR);
```

```
       endif
       if (not in_clock_skew(packet.timestamp,packet.usec)) then
               error_out(KRB_AP_ERR_SKEW);
       endif
       if (repeated(packet.timestamp,packet.usec,packet.s-address))
               then error_out(KRB_AP_ERR_REPEAT);
       endif
       if (packet.nonce is required or present) and
          (packet.nonce != expected-nonce) then
               error_out(KRB_AP_ERR_MODIFIED);
       endif

       for (ticket[n] in tickets that were forwarded) do
               save_for_later(ticket[n],key[n],principal[n],
                               server[n],times[n],flags[n]);
       return
```

A.20. KRB_ERROR generation

```
       /* assemble packet: */
       packet.pvno := protocol version; /* 5 */
       packet.msg-type := message type; /* KRB_ERROR */

       get system_time;
       packet.stime, packet.susec := system_time;
       packet.realm, packet.sname := server name;

       if (client time available) then
               packet.ctime, packet.cusec := client_time;
       endif
       packet.error-code := error code;
       if (client name available) then
               packet.cname, packet.crealm := client name;
       endif
       if (error text available) then
               packet.e-text := error text;
       endif
       if (error data available) then
               packet.e-data := error data;
       endif
```