Clifford Neuman
John Kohl
Theodore Ts'o
Ken Raeburn
Tom Yu
November 20, 2001
Expires 20 May, 2002

The Kerberos Network Authentication Service (V5)
draft-ietf-cat-kerberos-revisions-10

STATUS OF THIS MEMO

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC 2026. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at
http://www.ietf.org/shadow.html.

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.ietf.org (US East Coast), nic.nordu.net (Europe), ftp.isi.edu (US West Coast), or munnari.oz.au (Pacific Rim).

The distribution of this memo is unlimited. It is filed as draft-ietf-cat-kerberos-revisions-10.txt, and expires May 20, 2002. Please send comments to: ietf-krb-wg@anl.gov

ABSTRACT

This document provides an overview and specification of Version 5 of the Kerberos protocol, and updates RFC1510 to clarify aspects of the protocol and its intended use that require more detailed or clearer explanation than was provided in RFC1510. This document is intended to provide a detailed description of the protocol, suitable for implementation, together with descriptions of the appropriate use of protocol messages and fields within those messages.

This document is not intended to describe Kerberos to the end user, system administrator, or application developer. Higher level papers describing

Version 5 of the Kerberos system [NT94] and documenting version 4 [SNS88], are available elsewhere.

OVERVIEW

This INTERNET-DRAFT 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; they are more fully described in a paper available in IEEE communications [NT94] and earlier in the Kerberos portion of the Athena Technical Plan [MNSS87]. The protocols have been a proposed standard and are being considered for advancement for draft standard through the IETF standard process. Comments are encouraged on the presentation, but only minor refinements to the protocol as implemented or extensions that fit within current protocol framework will be considered at this time.

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 [NS78] and on modifications suggested by Denning and Sacco [DS81]. 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 5 of the Kerberos protocol (described in this document) has evolved from Version 4 based on new requirements and desires for features not available in Version 4. The design of Version 5 of the Kerberos protocol was led by Clifford Neuman and John Kohl with much input from the community. The development of the MIT reference implementation was led at MIT by John Kohl and Theodore T'so, with help and contributed code from many others. Since RFC1510 was issued, extensions and revisions to the protocol have been proposed by many individuals. Some of these proposals are reflected in this document. Where such changes involved significant effort, the document cites the contribution of the proposer.

Reference implementations of both version 4 and version 5 of Kerberos are publicly available and commercial implementations have been developed and

are widely used. Details on the differences between Kerberos Versions 4 and **5 can be found in [KNT92].**

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 assertions 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[1.1]. Kerberos performs authentication under these conditions as a trusted third-party authentication service by using conventional (shared secret key [1.2]) cryptography. Kerberos extensions (outside the scope of this document) can provide for the use of public key cryptography during certain phases of the authentication protocol [@RFCE: if PKINIT advances concurrently include reference to the RFC here]. Such extensions support Kerberos authentication for users registered with public key certification authorities and provide certain benefits of public key cryptography in situations where they are needed.

The basic Kerberos 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 a "ticket" for the server and 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.

Implementation of the basic protocol 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 directly or through the Generic Security Services Application Programming Interface, GSSAPI, described in separate document [ref to GSSAPI RFC]. These calls result in the transmission of the necessary messages to achieve authentication.

The Kerberos protocol consists of several sub-protocols (or exchanges). There are two basic 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 uses the TGT to authenticate itself to the TGS in the same manner as if it were contacting any other application server that requires Kerberos authentication. The reply is encrypted in the session key from the TGT. Though the protocol specification describes the AS and the TGS as separate servers, they are implemented in practice as different protocol entry points within a single Kerberos server.

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 application 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 originated with 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 contained in the ticket or the sub-session key found in the authenticator.

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). There is also a protocol for maintaining multiple copies of the Kerberos database. Neither of these protocols are described in this document.

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 prove its identity to servers in other realms[1.3]. 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 a 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.

The application server is ultimately responsible for accepting or rejecting authentication and should check the transited field. The application server may choose to rely on the KDC for the application server's realm to check the transited field. The application server's KDC will set the TRANSITED-POLICY-CHECKED flag in this case. The KDC's for intermediate

realms may also check the transited field as they issue ticket-granting-tickets for other realms, but they are encouraged not to do so. A client may request that the KDC's not check the transited field by setting the DISABLE-TRANSITED-CHECK flag. KDC's are encouraged but not required to honor this flag.

1.2. Choosing a principal with which to communicate

The Kerberos protocol provides the means for verifying (subject to the assumptions in 1.4) that the entity with which one communicates is the same entity that was registered with the KDC using the claimed identity (principal name). It is still necessary to determine whether that identity corresponds to the entity with which one intends to communicate.

When appropriate data has been exchanged in advance, this determination may be performed syntactically by the application based on the application protocol specification, information provided by the user, and configuration files. For example, the server principal name (including realm) for a telnet server might be derived from the user specified host name (from the telnet command line), the "host/" prefix specified in the application protocol specification, and a mapping to a Kerberos realm derived syntactically from the domain part of the specified hostname and information from the local Kerberos realms database.

One can also rely on trusted third parties to make this determination, but only when the data obtained from the third party is suitably integrity protected wile resident on the third party server and when transmitted. Thus, for example, one should not rely on an unprotected domain name system record to map a host alias to the primary name of a server, accepting the primary name as the party one intends to contact, since an attacker can modify the mapping and impersonate the party with which one intended to communicate.

1.3. Authorization

As an authentication service, Kerberos provides a means of verifying the identity of principals on a network. Authentication is usually useful primarily as a first step in the process of authorization, determining whether a client may use a service, which objects the client is allowed to access, and the type of access allowed for each. Kerberos does not, by itself, provide authorization. Possession of a client ticket for a service provides only for authentication of the client to that service, and in the absence of a separate authorization procedure, it should not be considered by an application as authorizing the use of that service.

Such separate authorization methods may be implemented as application specific access control functions and may utilize files on the application server, or on separately issued authorization credentials such as those

based on proxies [Neu93], or on other authorization services. Separately authenticated authorization credentials may be embedded in a tickets authorization data when encapsulated by the kdc-issued authorization data element.

Applications should not accept the mere issuance of a service ticket by the Kerberos server (even by a modified Kerberos server) as granting authority to use the service, since such applications may become vulnerable to the bypass of this authorization check in an environment if they interoperate with other KDCs or where other options for application authentication (e.g. the PKTAPP proposal) are provided.

1.4. 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 the protocols where an 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, but is typically on the order of 5 minutes. 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.

1.5. 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 named client or server entity that participates in a network communication, with one name that is considered canonical.

Principal identifier
: The canonical 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 may be derived from a password.

Server
: A particular Principal which provides a resource to network clients. The server is sometimes referred to as the Application Server.

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 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. With the excepttion of the ANONYMOUS and INVALID flags clients may ignore ticket flags that are not recognized.

2.1. Initial, pre-authenticated, and hardware 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 demonstrated knowledge of a client's secret key (e.g. a password-changing 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 additional 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) (see section 3.3.3.1).

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 short-lived 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 process of granting a proxy using the proxy and proxiable flags is used to provide credentials for use with specific services. Though conceptually also a proxy, user's wishing to delegate their identity for ANY purpose must

use the ticket forwarding mechanism described in the next section to forward a ticket granting ticket.

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 if requested by the client on initial authentication. By default, the client will request that it be set when requesting a ticket granting ticket, and reset when requesting any other ticket.

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[2.1]. When granting a proxy, the client must specify the new network address from which the proxy is to be used, or indicate that the proxy is to be issued for use from any address.

The PROXY flag is set in a ticket by the TGS when it issues a proxy ticket. Application servers may check this flag and at their option they may 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 a proxy where the service granted is 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 result can still be achieved if the user engages in the AS exchange specifying 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 a forwarded ticket 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 choose to process FORWARDED tickets differently than non-FORWARDED tickets.

2.7 Transited Policy Checking

In Kerberos, the application server is ultimately responsible for accepting or rejecting authentication and should check that only suitably trusted KDC's are relied upon to authenticate a principal. The transited field in the ticket identifies which KDC's were involved in the authentication process and an application server would normally check this field. While the end server ultimately decides whether authentication is valid, the KDC for the end server's realm may apply a realm specific policy for validating the transited field and accepting credentials for cross-realm authentication. When the KDC applies such checks and accepts such cross-realm authentication it will set the TRANSITED-POLICY-CHECKED flag in the service tickets it issues based on the cross-realm TGT. A client may request that the KDC's not check the transited field by setting the DISABLE-TRANSITED-CHECK flag. KDC's are encouraged but not required to honor this flag.

2.8 Anonymous Tickets

When policy allows, a KDC may issue anonymous tickets for the purpose of enabling encrypted communication between a client and server without identifying the client to the server. Such anonymous tickets are issued with a generic principal name configured on the KDC (e.g. "anonymous@") and will have the ANONYMOUS flag set. A server accepting such a ticket may assume that subsequent requests using the same ticket and session key originate from the same user. Requests with the same username but different tickets are likely to originate from different users. Users request anonymous ticket by setting the REQUEST-ANONYMOUS option in an AS or TGS request.

2.9. Other KDC options

There are three additional options which may be set in a client's request of the KDC.

2.9.1 Renewable-OK

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.

2.9.2 ENC-TKT-IN-SKEY

In its basic form the Kerberos protocol supports authentication in a client server setting and is not well suited to authentication in a peer-to-peer environment because the long term key of the user does not remain on the workstation after initial login. Authentication of such peers may be supported by Kerberos in its user-to-user variant. The ENC-TKT-IN-SKEY option supports user-to-user authentication by allowing the KDC to issue a service ticket encrypted using the session key from another ticket granting ticket issued to another user. The ENC-TKT-IN-SKEY option is honored only by the ticket-granting service. It indicates that the ticket to be issued for the end server is to be encrypted in the session key from the additional second 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 KRB_ERROR	5.4.2 5.9.1

The Authentication Service (AS) Exchange between the client and the Kerberos Authentication Server is initiated by a client when it wishes to obtain authentication credentials for a given server but currently holds no credentials. In its basic form, 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[3.1]. This exchange does not by itself provide any assurance of the the identity of the user[3.2].

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.

Without pre-authentication, 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 pre-authentication as described in section 3.1.1.

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 contains information which can be used to associate it with the message to which it replies. If suitable preauthentication has occurred, an optional checksum may be included in the KRB_ERROR message to prevent fabrication or modification of the KRB_ERROR message. When a checksum is not present, the lack of integrity protection precludes the ability to detect replays, fabrications, or modifications of the message, and the client must not depend on information in the KRB_ERROR message for security critical operations.

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 pre-authentication 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; whether the client requests an anonymous ticket; 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 non-renewable 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 the requested client principal named in the request is not known because it

doesn't exist in the KDC's principal database, then an error message with a KDC_ERR_C_PRINCIPAL_UNKNOWN is returned.

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 pre-authentication is required, but was not present in the request, an error message with the code KDC_ERR_PREAUTH_REQUIRED is returned and the PA-ETYPE-INFO pre-authentication field will be included in the KRB-ERROR message. If the server cannot accommodate an encryption type requested by the client, an error message with code KDC_ERR_ETYPE_NOSUPP is returned. Otherwise the KDC generates a 'random' session key[3.3].

When responding to an AS request, if there are multiple encryption keys registered for a client in the Kerberos database (or if the key registered supports multiple encryption types; e.g. DES3-CBC-SHA1 and DES3-CBC-SHA1-KD), then the etype field from the AS request is used by the KDC to select the encryption method to be used to protect the encrypted part of the KRB_AS_REP message which is sent to the client. If there is more than one supported strong encryption type in the etype list, the first valid etype for which an encryption key is available is used. The encryption method used to protect the encrypted part of the KRB_TGS_REP message is the keytype of the session key found in the ticket granting ticket presented in the KRB_TGS_REQ.

If the user's key was generated using an alternate string to key function than that used by the selected encryption type, information needed by the string to key function will be returned to the client in the padata field of the KRB_AS_REP message using the PA-PW-SALT, PA-AFS3-SALT, or similar pre-authentication typed values. This does not affect the encryption performed by the KDC since the key stored in the principal database already has the string to key transformation applied.

When the etype field is present in a KDC request, whether an AS or TGS request, the KDC will attempt to assign the type of the random session key from the list of methods in the etype field. The KDC will select the appropriate type using the list of methods provided together with information from the Kerberos database indicating acceptable encryption methods for the application server. The KDC will not issue tickets with a weak session key encryption type.

If the requested start time is absent, indicates a time in the past, or is within the window of acceptable clock skew for the KDC and the POSTDATE option has not been specified, then the start time of the ticket is set to the authentication server's current time. If it indicates a time in the future beyond the acceptable clock skew, 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 earlier of the requested endtime and a time determined by local policy, possibly determined using realm or principal specific factors. For example, the expiration time may 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 from the authentication server's database (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, ANONYMOUS. If the new ticket is post-dated (the start time is in the future), its INVALID flag will also be set.

If all of the above succeed, the server will encrypt ciphertext part of the ticket using the encryption key extracted from the server principal's record in the Kerberos database using the encryption type associated with the server principal's key (this choice is NOT affected by the etype field in

the request). It then 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 an acceptable encryption method requested in the etype field of the request, and sends the message 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, e-text, and optional e-cksum 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 (or are otherwise expected values), 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 for the host 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 for the host. 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 KRB_ERROR	5.5.2 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[3.4], 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 the ticket. 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. To use a ticket the client 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[3.5]. 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 [3.6].

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.

Unless the application server provides its own suitable means to protect against replay (for example, a challenge-response sequence initiated by the server after authentication, or use of a server-generated encryption subkey), the server must utilize a replay cache to remember any authenticator presented within the allowable clock skew. Careful analysis of the application protocol and implementation is recommended before eliminating this cache. The replay cache will store the server name, along with the client name, time and microsecond fields from the recently-seen authenticators and if a matching tuple is found, the KRB_AP_ERR_REPEAT error is returned [3.7]. If a server loses track of authenticators presented within the allowable clock skew, it must reject all requests until the clock

skew interval has passed, providing assurance that any lost or re-played authenticators will fall outside the allowable clock skew and can no longer be successfully replayed[3.8].

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.

If multiple servers (for example, different services on one machine, or a single service implemented on multiple machines) share a service principal (a practice we do not recommend in general, but acknowledge will be used in some cases), they should also share this replay cache, or the application protocol should be designed so as to eliminate the need for it. Note that

this applies to all of the services, if any of the application protocols does not have replay protection built in; an authenticator used with such a service could later be replayed to a different service with the same service principal but no replay protection, if the former doesn't record the authenticator information in the common replay cache.

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.

Passing these checks provides only authentication of the named principal; it does not imply authorization to use the named service. Applications must make a separate authorization decisions based upon the authenticated name of the user, the requested operation, local access control information such as that contained in a .k5login or .k5users file, and possibly a separate distributed authorization service.

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)[3.9]. 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[3.10] 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. 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 several alternatives. 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 session key from the ticket

and subkeys in the KRB_AP_REP message and the authenticator[3.11]. To mitigate the effect of failures in random number generation on the client it is strongly encouraged that any key derived by an application for subsequent use include the full key entropy derived from the KDC generated session key carried in the ticket. 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, but an example of how this might be done follows.

One way that an application may choose to negotiate a key to be used for subsequent integrity and privacy protection is for the client to propose a key in the subkey field of the authenticator. The server can then choose a key using the proposed key from the client as input, returning the new subkey in the subkey field of the application reply. This key could then be used for subsequent communication.

To make this example more concrete, if the communication patterns of an application dictates the use of encryption modes of operation incompatible with the encryption system used for the authenticator, then a key compatible with the required encryption system may be generated by either the client, the server, or collaboratively by both and exchanged using the subkey field. This generation might involve the use of a random number as a pre-key,

initially generated by either party, which could then be encrypted using the session key from the ticket, and the result exchanged and used for subsequent encryption. By encrypting the pre-key with the session key from the ticket, randomness from the KDC generated key is assured of being present in the negotiated key. Application developers must be careful however, to use a means of introducing this entropy that does not allow an attacker to learn the session key from the ticket if it learns the key generated and used for subsequent communication. The reader should note that this is only an example, and that an analysis of the particular cryptosystem to be used, must be made before deciding how to generate values for the subkey fields, and the key to be used for subsequent communication.

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 response 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 KRB_ERROR	5.4.2 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 sub-session 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 except when an optional checksum is included in the KRB_ERROR message, it is not possible 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 believed to be registered[3.12]. If the client knows the service principal name and realm and it 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 a Kerberos server for which the client possesses a ticket-granting ticket (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 between the client's realm and the requested realm server's realm).

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 or it may be generated from the name of the realm; as long as the secret keys exchanged by realms are kept secret, only denial of service results from using 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[3.13]. 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, using 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 or for a ticket granting server of an intermediate KDC to be contacted to obtain the requested ticket. The Kerberos database is queried to retrieve the record for the appropriate 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. If the requested server cannot be found in the TGS database, then a TGT for another trusted realm may be returned instead of a ticket for the service. This TGT is a referral mechanism to cause the client to retry the request to the realm of the TGT. These are the only cases where the response for 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 (endtime-starttime) 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, indicates a time in the past, or is within the window of acceptable clock skew for the KDC and the POSTDATE option has not been specified, then the start time of the ticket is set to the authentication server's current time. If it indicates a time in the future beyond the acceptable clock skew, 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 MAY-POSTDATE 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.

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, the server is registered in the realm of the KDC, and the RENEW option is requested, then the KDC will verify that the RENEWABLE flag is set in the ticket, that the INVALID flag is not set in the ticket, and that the renew_till time is still in the future. If the VALIDATE option is requested, 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, and the ticket passes the hotlist check described in the next section, the KDC will issue the appropriate new ticket.

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. Checking for revoked tickets

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 timestamps 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 to the KDC for the realm in which the server resides. 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. If TGT's have been issued for cross-realm authentication, use of the cross-realm TGT will not be affected unless the hot-list is propagated to the KDC's for the

realms for which such cross-realm tickets were issued.

3.3.3.2. 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 end-points, 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[18]. 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[19]. 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. The server name returned in the reply is the true principal name of the service. 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 collision-proof 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 occurred).

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 a 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 collision-proof keyed checksum, and if it is not, a KRB_AP_ERR_INAPP_CKSUM error is generated. If the sender's address was included in the control information, 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 (sent or received[20]) 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 time-stamp 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 occurred). 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. If the sender's address was included in the control information, 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 time-stamp 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 r-address fields, are placed in the encrypted part of the KRB_CRED message which is then encrypted under an encryption key previously exchanged in the KRB_AP exchange (usually the last key negotiated via subkeys, or the session key if no negotiation has occurred).

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 any principals to be authenticated[4.1] using such secret keys. The keying material in the database must be protected so that they are only accessible to the Kerberos server and administrative functions specifically authorized to access such material. Specific implementations may handle the storage of keying material separate from the Kerberos database (e.g. in hardware) or by encrypting the keying material before placing it in the Kerberos database. Some implementations might provide a means for using long term secret keys, but not for retrieving them from the Kerberos database.

4.1. Database contents

A database entry will typically contain the following fields, though in some instances a KDC may obtain these values through other means:

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 combines 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 [LGDSR87]), to identify whether a principal can play the role of a client or server or both, to note whether a server is appropriately trusted to receive credentials delegated by a client, or to identify the 'string to key' conversion algorithm used for a principal's key[4.2]. 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

NOTE: We are continuing to work on changes to message format extensibility as discussed at the London meeting. We believe the general form discussed in London will continue to be a useful strategy for pursuing this goal. We expect to have additional information by the Salt Lake City meeting. TODO: TypedData needs to be looked at carefully, particularly with regard to TD-APP-DEFINED-ERROR, etc. Some significant changes from 1510 to here have been written up; more proofreading is needed. - tlyu

The Kerberos protocol is defined here in terms of Abstract Syntax Notation One (ASN.1), which provides a syntax for specifying both the abstract layout of protocol messages as well as their encodings. Implementors not utilizing an existing ASN.1 compiler or support library are cautioned to thoroughly understand the actual ASN.1 specification to ensure correct implementation behavior, as there is more complexity in the notation than is immediately obvious, and some tutorials and guides to ASN.1 are misleading or erroneous.

Note that in several places, there have been changes here from RFC 1510 that change the abstract types. This is in part to address widespread assumptions that various implementations have made, in some cases unintentionally violating the ASN.1 standard in various ways. These will be clearly flagged when they occur. The changes to the abstract types can cause incompatible encodings to be emitted when certain encoding rules, e.g. the Packed Encoding Rules (PER) are used. This should not be relevant for Kerberos, since Kerberos explicitly specifies the use of the Distinguished Encoding Rules (DER). This might be an issue for protocols wishing to use Kerberos types with other encoding rules. (This practice is not recommended.) With very few exceptions (most notably the usages of BIT STRING), the encodings emitted by the DER, which are the only encodings permitted by this document and by RFC 1510, remain identical.

The type definitions in this section assume an ASN.1 module definition of the following form:

```
Kerberos5 {
    iso (1), org(3), dod(6), internet(1), security(5), kerberosV5(2)
} DEFINITIONS ::= BEGIN

-- rest of definitions here

END
```

This specifies an explicit non-automatic tagging for the ASN.1 type definitions.

Note that in some other publications [RFC1510] [RFC1964], the "dod" portion of the object identifier is erroneously specified as having the value "5".

Note that elsewhere in this document, nomenclature for various message types is inconsistent, but seems to largely follow C language conventions, including use of underscore (_) characters and all-caps spelling of names intended to be numeric constants. Also, in some places, identifiers (especially ones refering to constants) are written in all-caps in order to distinguish them from surrounding explanatory text.

The ASN.1 notation does not permit underscores in identifiers, so in actual ASN.1 definitions, underscores are replaced with hyphens (-). Additionally, structure member names and defined values in ASN.1 must begin with a lowercase letter, while type names must begin with an uppercase letter.

5.1. Specific Compatibility Notes on ASN.1

For compatibility purposes, implementors should heed the following specific notes regarding the use of ASN.1 in Kerberos. These notes do not describe a non-standard usage of ASN.1, but rather some historical quirks and non-compliance of various implementations, as well as historical ambiguities, which, while being valid ASN.1, can lead to confusion during implementation.

5.1.1. ASN.1 Distinguished Encoding Rules

The encoding of Kerberos protocol messages shall obey the Distinguished Encoding Rules (DER) of ASN.1 as described in X.690 (1997). Some implementations (believed to be primarly ones derived from DCE 1.1 and earlier) are known to use the more general Basic Encoding Rules (BER); in particular, these implementations send indefinite encodings of lengths. Implementations may accept such encodings in the interests of backwards compatibility, though implementors are warned that decoding fully-general BER is fraught with peril.

5.1.2. Optional Fields in ASN.1 Sequences

Some implementations behave as if certain default values are equivalent to omission of an optional value. Implementations should handle this case gracefully. For example, the seq-number field in an Authenticator is optional, but some implementations use an internal value of zero to indicate that the field is to be omitted upon encoding. [While it is possible to use the DEFAULT qualifier for the ASN.1 notation of a SEQUENCE member in order to mandate this behavior, the result would be that the member would be mandatory to omit if the value intended is that specified by the DEFAULT keyword. This limits the possible semantics of the protocol.]

5.1.3. Zero-length SEQUENCE Types

There are places in the protocol where a message contains a SEQUENCE OF type as an optional member, or a SEQUENCE type where all members are optional. This can result in an encoding that contains an zero-length SEQUENCE or SEQUENCE OF encoding. In general, implementations should not send zero-length SEQUENCE OF or SEQUENCE encodings that are marked OPTIONAL, but should accept them. [XXX there may be cases where an empty SEQUENCE type has useful semantics, though]

5.1.4. Unrecognized Tag Numbers

Future revisions to this protocol may include new message types with different APPLICATION class tag numbers. Such revisions should protect older implementations by only sending the message types to parties that are known to understand them, e.g. by means of a flag bit set by the receiver in a preceding request. In the interest of robust error handling, implementations

should gracefully handle receiving a message with an unrecognized tag anyway, and return an error message if appropriate.

5.1.5. Tag Numbers Greater Than 30

A naive implementation of a DER ASN.1 decoder may experience problems with ASN.1 tag numbers greater than 30, due such tag numbers being encoded using more than one byte. Future revisions of this protocol may utilize tag numbers greater than 30, and implementations should be prepared to gracefully return an error, if appropriate, if they do not recognize the tag.

5.2. Basic Kerberos Types

This section defines a number of basic types that are potentially used in multiple Kerberos protocol messages.

5.2.1. KerberosString

[XXX The following paragraphs may need some editing, or maybe they want to live in a footnote]

The original specification of the Kerberos protocol in RFC 1510 uses GeneralString in numerous places for human-readable string data. Historical implementations of Kerberos cannot utilize the full power of GeneralString. This ASN.1 type requires the use of designation and invocation escape sequences as specified in ISO 2022 to switch character sets, and the default character set that is designated for G0 is basically US ASCII, which mostly works. In practice, many implementations end up treating GeneralStrings as if they were strings of whatever character set the implementation defaults to, without regard for correct usage of character set designation escape sequences.

Also, DER prohibits the invocation of character sets into any but the G0 and C0 sets, which seems to outright prohibit the encoding of characters with the high bit set. Unfortunately, this seems to have the side effect of prohibiting the transmission of Latin-1 characters or any other characters that belong to a 96-character set, since it is prohibited to invoke them into G0. Some inconclusive discussion has taken place within the ASN.1 community on this subject. For now, we must assume that the ASN.1 specification of GeneralString as currently published is fundamentally flawed in several ways.

One method of resolving these myriad difficulties is to constrain the use of GeneralString to only include IA5String, which is essentially the US-ASCII. US-ASCII control characters should in general not be used in KerberosString, except for cases such as newlines in lengthy error messages.

The new (since RFC 1510) type KerberosString, defined below, is a CHOICE

containing a GeneralString that is constrained to only contain characters in IA5String (which are US-ASCII). Note that the ASN.1 standard does not permit the use of escape sequences to change the character sets while encoding an IA5String.

```
KerberosString ::= CHOICE {
    general GeneralString (IA5String),
    ...
}
```

This CHOICE is extensible, so that when an interoperable solution for internationalization is chosen, it will be easier to specify the changed types. In the future, changes to this protocol that allow for extensions to this CHOICE will be specified so that the transmitting party has some way of knowing whether the receiving party can accept the chosen alternative of the CHOICE.

Implementations may choose to accept GeneralString values that contain characters other than those permitted by IA5String, but they should be aware that character set designation codes will likely be absent, and that the encoding should probably be treated as locale-specific in almost every way. Implementations may also choose to emit GeneralString values that are beyond those permitted by IA5String, but should be aware that doing so is extraordinarily risky from an interoperability perspective.

Some existing implementations use GeneralString to encode unescaped locale-specific characters. This is in violation of the ASN.1 standard. Most of these implementations encode US-ASCII in the left-hand half, so as long the implementation transmits only US-ASCII, the ASN.1 standard is not violated in this regard. As soon as such an implementation encodes unescaped locale-specific characters with the high bit set, it violates the ASN.1 standard.

Other implementations have been known to use GeneralString to contain a UTF-8 encoding. This also violates the ASN.1 standard, since UTF-8 is a different encoding, not a 94 or 96 character "G" set as defined by ISO 2022. It is believed that these implementations do not even use the ISO 2022 escape sequence to change the character encoding. Even if implementations were to announce the change of encoding by using that escape sequence, the ASN.1 standard prohibits the use of any escape sequences other than those used to designate/invoke "G" or "C" sets allowed by GeneralString.

Future revisions to this protocol will almost certainly allow for a more interoperable representation of principal names, probably including UTF8String.

Note that both applying a new constraint to a previously unconstrained type and replacing a type with a CHOICE containing that type constitute creations

of new ASN.1 types. In the case here, the change here does not result in a changed encoding under DER. Also, note that various text in the ASN.1 standard actually suggests the strategy of replacing a type with a CHOICE containing that type for certain deprecated types, even though this creates a new type.

5.2.2. Realm and PrincipalName

```
Realm ::=           KerberosString
PrincipalName ::=   SEQUENCE {
                    name-type[0]     Int32,
                    name-string[1]   SEQUENCE OF KerberosString
}
```

Kerberos realm names are encoded as KerberosStrings. 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 KerberosString. Taken together, a PrincipalName and a Realm form a principal identifier. Most PrincipalNames will have only a few components (typically one or two).

5.2.3. KerberosTime

```
KerberosTime ::=   GeneralizedTime
                   -- with no fractional seconds
```

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

5.2.4. Constrained Integer types

Some integer members of types should be constrained to values representable in 32 bits, for compatibility with reasonable implementation limits.

```
Int32 ::= INTEGER (-2147483648..2147483647)
          -- signed values representable in 32 bits

UInt32 :: = INTEGER (0..4294967295)
             -- unsigned 32 bit values

Microseconds ::= INTEGER (0..99999)
                 -- microseconds
```

While this results in changes to the abstract types from the RFC 1510 version, the encoding in DER should be unaltered. Historical implementations were typically limited to 32-bit integer values anyway, and assigned numbers should fall in the space of integer values representable in 32 bits in order to promote interoperability anyway.

There are some members of messages types that are still defined as unconstrained INTEGER types, but many of these have a (non-ASN.1) constraint applied in the descriptive text. There are specific cases where more discussion needs to occur regarding possible constraints, such as for the nonce fields in various messages.

5.2.5. HostAddress and HostAddresses

```
HostAddress ::=     SEQUENCE  {
                    addr-type[0]            Int32,
                    address[1]              OCTET STRING
}

HostAddresses ::=   SEQUENCE OF HostAddress
```

The host address 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.

5.2.6. AuthorizationData

```
AuthorizationData ::=   SEQUENCE OF SEQUENCE {
                        ad-type[0]              Int32,
                        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.

Each sequence of type and data is referred to as an authorization element. Elements may be application specific, however, there is a common set of recursive elements that should be understood by all implementations. These elements contain other elements embedded within them, and the interpretation of the encapsulating element determines which of the embedded elements must be interpreted, and which may be ignored. Definitions for these common elements may be found in Appendix B.

5.2.7. PA-DATA

Historically, PA-DATA have been known as "pre-authentication data", meaning that they were used to augment the initial authentication with the KDC. Since that time, they have also been used as a typed hole with which to extend protocol exchanges with the KDC.

```
PA-DATA ::=        SEQUENCE {
                   padata-type[1]     Int32,
                   padata-value[2]    OCTET STRING
                                      -- might be encoded AP-REQ
}
```

padata-type
     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.
padata-value
     Usually contains the DER encoding of another type; the padata-type field identifies which type is encoded here.

padata-type	name	contents of padata-value
1	pa-tgs-req	DER encoding of AP-REQ
2	pa-enc-timestamp	DER encoding of PA-ENC-TIMESTAMP
3	pa-pw-salt	salt (not ASN.1 encoded)
10	pa-etype-info	DER encoding of PA-ETYPE-INFO
20	pa-use-specified-kvno	DER encoding of INTEGER

[XXX -- the following paragraph needs discussion, as does the general concept of authenticating the cleartext pieces of the protocol]

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. When this field is used to authenticate or pre-authenticate a request, it should contain a keyed checksum over the KDC-REQ-BODY to bind the pre-authentication data to rest of the request. The KDC, as a matter of policy, may decide whether to honor a KDC-REQ which includes any pre-authentication data that does not contain the checksum field.

It may also be used by the client to specify the version of a key that is being used for accompanying preauthentication, and/or which should be used to encrypt the reply from the KDC. [XXX the following paragraph should apply perhaps to PA-DATA in general]

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 token cards with Kerberos. The details of such extensions are specified in separate documents. See [Pat92] for additional uses of this field.

5.2.7.1. PA-TGS-REQ

In the case of requests for additional tickets (KRB_TGS_REQ), padata-value will contain an encoded AP-REQ. The checksum in the authenticator (which must be collision-proof) is to be computed over the KDC-REQ-BODY encoding.

5.2.7.2. Encrypted Timestamp Pre-authentication

There are pre-authentication types that may be used to pre-authenticate a client by means of an encrypted timestamp. The original PA-ENC-TIMESTAMP does not contain a checksum of the KDC-REQ-BODY, while the PA-ENC-TIMESTAMP2 does.

```
PA-ENC-TIMESTAMP ::= EncryptedData -- encrypted PA-ENC-TS-ENC

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

-- XXX maybe remove ENC-TIMESTAMP2 for now?

PA-ENC-TIMESTAMP2 ::= EncryptedData -- encrypted PA-ENC-TS2-ENC

PA-ENC-TS2-ENC   ::= SEQUENCE {
       patimestamp[0]              KerberosTime, -- client's time
       pausec[1]                   Microseconds OPTIONAL,
       pachecksum[2]               Checksum OPTIONAL
```

```
                                    -- keyed checksum of KDC-REQ-BODY
}
```

Patimestamp contains the client's time, and pausec contains 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 or PA-ENC-TS2-ENC encoding, encrypted using the client's secret key.

This preauthentication type was not present in RFC 1510, but many implementations support it.

5.2.7.3. PA-PW-SALT

The padata-value for this preauthentication type contains the salt for the string-to-key to be used by the client to obtain the key for decrypting the encrypted part of an AS-REP message. Unfortunately, for historical reasons, the character set to be used is unspecified and probably locale-specific.

This preauthentication type was not present in RFC 1510, but many implementations support it. It is necessary in any case where the salt for the string-to-key algorithm is not the default.

In the trivial example, a zero-length salt string is very commonplace for realms that have converted their principal databases from Kerberos 4.

5.2.7.4. PA-ETYPE-INFO

The ETYPE-INFO preauthentication type is sent by the KDC in a KRB-ERROR indicating a requirement for additional preauthentication. It is usually used to notify a client of which key to use for the encryption of an encrypted timestamp for the purposes of sending a PA-ENC-TIMESTAMP preauthentication value.

```
ETYPE-INFO-ENTRY ::= SEQUENCE {
        etype[0]        INTEGER,
        salt[1]         OCTET STRING OPTIONAL
}

ETYPE-INFO ::= SEQUENCE OF ETYPE-INFO-ENTRY
```

The salt, like that of PA-PW-SALT, is also completely unspecified with respect to character set and is probably locale-specific.

[XXX -- not clear whether ETYPE-INFO or PW-SALT should take precedence if they conflict]

This preauthentication type was not present in RFC 1510, but many implementations that support encrypted timestamps for preauthentication need

to support ETYPE-INFO as well.

5.2.7.5. PA-USE-SPECIFIED-KVNO

The KDC should only accept and abide by the value of the use-specified-kvno preauthentication data field when the specified key is still valid and until use of a new key is confirmed. This situation is likely to occur primarily during the period during which an updated key is propagating to other KDC's in a realm.

5.2.8. KerberosFlags

For several message types, a specific constrained bit string type, KerberosFlags, is used.

KerberosFlags ::= BIT STRING (SIZE (32..MAX))

Compatibility note: the following paragraphs describe a change from the RFC1510 description of bit strings that would result in incompatility in the case of an implementation that strictly conformed to ASN.1 DER and RFC1510.

ASN.1 bit strings have multiple uses. The simplest use of a bit string is to contain a vector of bits, with no particular meaning attached to individual bits. This vector of bits is not necessarily a multiple of eight bits long. The use in Kerberos of a bit string as a compact boolean vector wherein each element has a distinct meaning poses some problems. The natural notation for a compact boolean vector is the ASN.1 "NamedBit" notation, and the DER require that encodings of a bit string using "NamedBit" notation exclude any trailing zero bits. This truncation is easy to neglect, especially given C language implementations that may naturally choose to store boolean vectors as 32 bit integers.

For example, if the notation for KDCOptions were to include the "NamedBit" notation, as in RFC 1510, and a KDCOptions value to be encoded had only the "forwardable" (bit number one) bit set, the DER encoding must only include two bits: the first reserved bit ("reserved", bit number zero, value zero) and the one-valued bit (bit number one) for "forwardable".

Most existing implementations of Kerberos unconditionally send 32 bits on the wire when encoding bit strings used as boolean vectors. This behavior violates the ASN.1 syntax used for flag values in RFC 1510, but occurs on such a widely installed base that the protocol description is being modified to accomodate it.

Consequently, this document removes the "NamedBit" notations for individual bits, relegating them to comments. The size constraint on the KerberosFlags type requires that at least 32 bits be encoded at all times, though a lenient implementation may choose to accept fewer than 32 bits and to treat the missing bits as set to zero.

Currently, no uses of KerberosFlags specify more than 32 bits worth of flags, although future revisions of this document may do so. When more than **32 bits are to be transmitted in a KerberosFlags value, future revisions to** this document will likely specify that the smallest number of bits needed to encode the highest-numbered one-valued bit should be sent. This is somewhat similar to the DER encoding of a bit string that is declared with the "NamedBit" notation.

5.2.9. Cryptosystem-related Types

Many Kerberos protocol messages contain an EncryptedData as a container for arbitrary encrypted data, which is often the encrypted encoding of another data type. Fields within EncryptedData assist the recipient in selecting a key with which to decrypt the enclosed data.

```
EncryptedData ::=   SEQUENCE {
                    etype[0]     Int32, -- 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 in section 6.

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 EncryptionKey type is the means by which cryptographic keys used for encryption are transfered.

```
EncryptionKey ::=   SEQUENCE {
                    keytype[0]    Int32, -- actually encryption type
                    keyvalue[1]   OCTET STRING
}
```

keytype
: This field specifies the encryption type of the encryption key that follows in the keyvalue field. While its name is "keytype", it actually specifies an encryption type. Previously, multiple cryptosystems that performed encryption differently but were capable of using keys with the same characteristics were permitted to share an assigned number to designate the type of key; this usage is now deprecated.

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.

Messages containing cleartext data to be authenticated will usually do so by using a member of type Checksum. Most instances of Checksum use a keyed hash, though exceptions will be noted.

```
Checksum ::=   SEQUENCE {
               cksumtype[0]   Int32,
               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 in section 6. Negative values for the checksum type are reserved for local use. All non-negative values are reserved for officially assigned type fields and interpretations.

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 --EncTicketPart
}

-- 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]            Int32, -- must be registered
                        contents[1]           OCTET STRING
}

TicketFlags ::= KerberosFlags
                  -- 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),
                  -- transited-policy-checked(12),
                  -- ok-as-delegate(13)
                  -- anonymous(14)
```

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 all components of the name part of the server's

identity, including those parts that identify a specific instance of a service.

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). [XXX X.690 ref and notes on pitfalls?] 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.

		This flag indicates that this ticket has been postdated. The end-service

6	postdated	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.
		This flag indicates that the KDC for the realm has checked the transited field against a realm defined policy for trusted certifiers. If this flag is reset (0), then the application

		server must check the transited field itself, and if unable to do so it must reject the authentication. If the flag

12	transited-policy-checked	is set (1) then the application server may skip its own validation of the transited field, relying on the validation performed by the KDC. At its option the application server may still apply its own validation based on a separate policy for acceptance. This flag is new since RFC 1510.
13	ok-as-delegate	This flag indicates that the server (not the client) specified in the ticket has been determined by policy of the realm to be a suitable recipient of delegation. A client can use the presence of this flag to help it make a decision whether to delegate credentials (either grant a proxy or a forwarded ticket granting ticket) to this server. The client is free to ignore the value of this flag. When setting this flag, an administrator should consider the Security and placement of the server on which the service will run, as well as whether the service requires the use of delegated credentials. This flag is new since RFC 1510.
14	anonymous	This flag indicates that the principal named in the ticket is a generic principal for the realm and does not identify the individual using the ticket. The purpose of the ticket is only to securely distribute a session key, and not to identify the user. Subsequent requests using the same ticket and session may be considered as originating from the same user, but requests with the same username but a different ticket are likely to originate from different users.

This flag is new since RFC 1510.

15-31 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 3.3.3.2 for details on how this field encodes the traversed realms. When the names of CA's are to be embedded in the transited field (as specified for some extensions to the protocol), the X.500 names of the CA's should be mapped into items in the transited field using the mapping defined by RFC2253.

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[24].

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. Experience has shown that the name of this field is confusing, and that a better name for this field would be restrictions. Unfortunately, it is not possible to change the name of this field at this time.

This field contains restrictions on any authority obtained on the basis of authentication using the ticket. It is possible for any principal in posession of credentials to add entries to the authorization data field since these entries further restrict what can be done with the ticket. Such additions can be made by specifying the additional entries when a new ticket is obtained during the TGS exchange, or they may be added during chained delegation using the authorization data field of the authenticator.

Because entries may be added to this field by the holder of credentials, except when an entry is separately authenticated by encapsulation in the kdc-issued element, it is not allowable for the presence of an entry in the authorization data field of a ticket to amplify the privileges one would obtain from using a ticket.

The data in this field may be specific to the end service; 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 sub-fields, 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.

A separate service providing authorization or certifying group membership may be built using the authorization-data field. In this case, the entity granting authorization (not the authorized entity), may obtain a ticket in its own name (e.g. the ticket is issued in the name of a privilege server), and this entity adds restrictions on its own authority and delegates the restricted authority through a proxy to the client. The client would then present this authorization credential to the application server separately from the authentication exchange. Alternatively, such authorization credentials may be embedded in the ticket authenticating the authorized entity, when the authorization is separately authenticated using the kdc-issued authorization data element (see B.4).

Similarly, 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 [Neu93] 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]                    Microseconds,
                ctime[5]                    KerberosTime,
                subkey[6]                   EncryptionKey OPTIONAL,
                seq-number[7]               UInt32 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. Sequence numbers fall in the range of 0 through 2^32 - 1 and wrap to zero following the value 2^32 - 1.

  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 the 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
}

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 Int32,
                                            -- EncryptionType,
                                            -- in preference order
                     addresses[9]           HostAddresses OPTIONAL,
                enc-authorization-data[10] EncryptedData OPTIONAL,
                                            -- Encrypted AuthorizationData
                                            -- encoding
                     additional-tickets[11] SEQUENCE OF Ticket OPTIONAL
}
```

```
KDCOptions ::=   KerberosFlags
                  -- 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),
                  -- unused12(12),
                  -- unused13(13),
                  -- requestanonymous(14),
                  -- canonicalize(15),
                  -- disable-transited-check(26),
                  -- renewable-ok(27),
                  -- enc-tkt-in-skey(28),
                  -- renew(30),
                  -- validate(31)
```

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
: Contains pre-authentication data. Requests for additional tickets (KRB_TGS_REQ) must contain a padata of PA-TGS-REQ.

  The padata (pre-authentication data) field contains a sequence of authentication information which may be needed before credentials can be issued or decrypted. In most requests for initial authentication (KRB_AS_REQ) and most replies (KDC-REP), the padata field will be left out.

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:

Bits	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	RESERVED	Reserved for PK-Cross
10-13	UNUSED	These options are presently unused.
14	REQUEST-ANONYMOUS	The REQUEST-ANONYMOUS option indicates that the ticket to be issued is not to identify the user to which it was issued. Instead, the principal identifier is to be generic, as specified by the policy of the realm (e.g. usually anonymous@realm). The purpose of the ticket is only to securely distribute a session key, and not to identify the user. The ANONYMOUS flag on the ticket to be returned should be set. If the local realms policy does not permit anonymous credentials, the request is to be rejected. This flag is new since RFC 1510
15	CANONICALIZE	The CANONICALIZE option indicates that the client will accept the return of a true server name instead of the name specified in the request. In addition the client will be able to process any TGT referrals that will direct the client to another realm to locate the requested server. If a KDC does not support name- canonicalization, the option is ignored and the appropriate KDC_ERR_C_PRINCIPAL_UNKNOWN or KDC_ERR_S_PRINCIPAL_UNKNOWN error is returned. [JBrezak] This flag is new since RFC 1510

16-25	RESERVED	Reserved for future use.
		By default the KDC will check the transited field of a ticket-granting-ticket against the policy of the local realm before it will issue derivative tickets based

26	DISABLE-TRANSITED-CHECK	on the ticket granting ticket. If this flag is set in the request, checking of the transited field is disabled. Tickets issued without the performance of this check will be noted by the reset (0) value of the TRANSITED-POLICY-CHECKED flag, indicating to the application server that the tranisted field must be checked locally. KDC's are encouraged but not required to honor the DISABLE-TRANSITED-CHECK option. This flag is new since RFC 1510
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. If the CANONICALIZE option is set, the realm is used as a hint to the KDC for its database lookup.

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. If this field is omitted then the KDC should use the current time instead.

till

This field contains the expiration date requested by the client in a ticket request. [XXX This was optional in kerberos-revisions, but required in 1510. we should make it required and specify semantics for 19700101000000Z] It is optional and if omitted the requested ticket is to have the maximum endtime permitted according to KDC policy for the parties to the authentication exchange as limited by expiration date of the ticket granting ticket or other preauthentication credentials.

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 generated randomly, but if the correct time is known, it may suffice[25].

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. When the ENC-TKT-IN-SKEY option is used for user-to-user authentication, this addional ticket may be a TGT issued by the local realm or an inter-realm TGT issued for the current KDC's realm by a remote KDC. 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 tag number 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 or EncTGSReoOart
}
```



```
EncASRepPart ::=    [APPLICATION 25] EncKDCRepPart -- note [27]
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
}
LastReq ::=   SEQUENCE OF SEQUENCE {
               lr-type[0]              Int32,
               lr-value[1]             KerberosTime
}
```

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.

  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).

If it is (6), then the lr-value subfield is the time when the password will expire.

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.

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 expiration 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
                                                -- Authenticator from 5.3.2
}
```

```
APOptions ::=   KerberosFlags
                  -- 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-key	The 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-required	The 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 time- stamp. 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
}
```

```
EncAPRepPart ::=   [APPLICATION 27] SEQUENCE { -- note [29]
                   ctime[0]                         KerberosTime,
                   cusec[1]                         Microseconds,
                   subkey[2]                        EncryptionKey OPTIONAL,
                   seq-number[3]                    UInt32 OPTIONAL
}
```


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.

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

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).

There are two KRB_SAFE messages; the KRB-SAFE message is the one specified in RFC 1510. The KRB-SAFE2 message is new with this document, and shares a

number of fields with the old KRB-SAFE message.

5.6.1. KRB_SAFE definition

The KRB_SAFE message contains user data along with a collision-proof checksum keyed with the last encryption key negotiated via subkeys, or the session key if no negotiation has occurred. 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]                         Microseconds OPTIONAL,
                    seq-number[3]                   UInt32 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 or KRB_SAFE2, respectively, for the KRB-SAFE and KRB-SAFE2 messages.

safe-body
: This field is a placeholder for the body of the KRB-SAFE message.

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 sequence. First, the cksum is set to a type zero, zero-length value and the checksum is computed over the encoding of the KRB-SAFE sequence, then the checksum is set to the result of that computation, and finally the KRB-SAFE sequence is encoded again. This method, while different than the one specified in RFC 1510, corresponds to existing practice.

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
     Sender's address.

     This field specifies the address in use by the sender of the message.
     It may be omitted if not required by the application protocol.
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
}
```



```
EncKrbPrivPart ::=  [APPLICATION 28] SEQUENCE { --note [31]
                    user-data[0]        OCTET STRING,
                    timestamp[1]        KerberosTime OPTIONAL,
                    usec[2]             Microseconds OPTIONAL,
                    seq-number[3]       UInt32 OPTIONAL,
                    s-address[4]        HostAddress, -- sender's addr
                    r-address[5]        HostAddress OPTIONAL -- recip's addr
}
```

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[32]. 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 encrypted under an encryption key previously exchanged or transferred alongside the KRB_CRED message. 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
}

EncKrbCredPart   ::= [APPLICATION 29]   SEQUENCE {
                 ticket-info[0]         SEQUENCE OF KrbCredInfo,
                 nonce[1]               INTEGER OPTIONAL,
                 timestamp[2]           KerberosTime OPTIONAL,
                 usec[3]                Microseconds 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 message 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 only optionally integrity protected, it is quite possible for an intruder to synthesize or modify such a message. In particular, this means that unless appropriate integrity protection mechanisms have been applied to the KRB_ERROR message, 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]                        Microseconds OPTIONAL,
                stime[4]                        KerberosTime,
                susec[5]                        Microseconds,
                error-code[6]                   Int32,
                crealm[7]                       Realm OPTIONAL,
                cname[8]                        PrincipalName OPTIONAL,
                realm[9]                        Realm, -- Correct realm
                sname[10]                       PrincipalName, -- Correct name
                e-text[11]                      KerberosString 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 999999. 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 present, this field will contain the encoding of a sequence of TypedData (TYPED-DATA below), unless the errorcode is KDC_ERR_PREAUTH_REQUIRED, in which case it will contain the encoding of a sequence of of padata fields (METHOD-DATA below), each corresponding to an acceptable pre-authentication method and optionally containing data for the method:

```
TYPED-DATA   ::=   SEQUENCE of TypedData
METHOD-DATA  ::=   SEQUENCE of PA-DATA

TypedData ::=   SEQUENCE {
                    data-type[0]   Int32,
                    data-value[1]  OCTET STRING OPTIONAL
}
```

Note that the padata-type field in the PA-DATA structure and the data-type field in the TypedData structure share a common range of allocated values which are coordinated to avoid conflicts. One Kerberos error message, KDC_ERR_PREAUTH_REQUIRED, embeds elements of type PA-DATA, while all other error messages embed TypedData.

While preauthentication methods of type PA-DATA should be encapsulated within a TypedData element of type TD-PADATA, for compatibility with old clients, the KDC should include PA-DATA types below 22 directly as method-data. All new implementations interpreting the METHOD-DATA field for the KDC_ERR_PREAUTH_REQUIRED message must accept a type of TD-PADATA, extract the typed data field and interpret the use any elements encapsulated in the TD-PADATA elements as if they were present

in the METHOD-DATA sequence.

Unless otherwise specified, unrecognized TypedData elements within the KRB-ERROR message MAY be ignored by implementations that do not support them. Note that all TypedData MAY be bound to the KRB-ERROR message via the checksum field.

An application may use the TD-APP-DEFINED-ERROR typed data type for data carried in a Kerberos error message that is specific to the application. TD-APP-SPECIFIC must set the data-type value of TypedData to TD-APP-SPECIFIC and the data-value field to

```
   AppSpecificTypedData as follows:
       AppSpecificTypedData ::= SEQUENCE {
               oid[0]           OPTIONAL OBJECT IDENTIFIER,
                                -- identifies the application
               data-value[1]    OCTET STRING
                                -- application
                                -- specific data
       }
```


- The TD-REQ-NONCE TypedData MAY be used to bind a KRB-ERROR to a KDC-REQ. The data-value is an INTEGER that is equivalent to the nonce in a KDC-REQ.

- The TD-REQ-SEQ TypedData MAY be used for binding a KRB-ERROR to the sequence number from an authenticator. The data-value is an INTEGER, and it is identical to sequence number sent in the authenticator.

- The data-value for TD-KRB-PRINCIPAL is the Kerberos-defined PrincipalName. The data-value for TD-KRB-REALM is the Kerberos-defined Realm. These TypedData types MAY be used to indicate principal and realm name when appropriate.

5.10. Application Tag Numbers

The following table lists the application class tag numbers used by various data types defined in this section.

Tag Number(s)	Type Name	Comments
0		unused
1	Ticket	
2	Authenticator	

3	EncTicketPart
4-10	unused
10	AS-REQ
11	AS-REP
12	TGS-REQ
13	TGS-REP
14	AP-REQ
15	AP-REP
16	TGT-REQ
17-19	unused
20	KRB-SAFE
21	KRB-PRIV
22	KRB-PRIV
23-24	unused
25	EncASRepPart
26	EncTGSRepPart

27	EncApRepPart
28	EncKrbPrivPart
29	EncKrbCredPart
30	KRB-ERROR

6. Encryption and Checksum Specifications

Work is still needed on this section.

* Re-synchronize the key usage value list with any changes Tom makes to the message definitions. KRB-ERROR checksum, for example, and any new message types.
* More talking with cryptographers about the "combine-keys" function in the simplified profile. I've been talking a little with Uri Blumenthal, but he hasn't had a lot of time

for this.
* Test vections need to go into an appendix. Submitted, appendix letter not yet assigned; I'm using "Z" in this text. Appendix letter also not yet assigned for deprecated checksum algorithms, assuming "Y".
* More detailed list of differences from RFC 1510, to update the "Significant changes" appendix.
* Are sections 6.2 and 6.3 what we want to recommend for external use in section 6.7, or just a subset?
* Fix up reference to Bellovin paper.
* Fix anything marked with "@@".

See end notes for other issues.

-- Ken 2001-11-20

The Kerberos protocols described in this document are designed to encrypt blocks of arbitrary sizes, using stream encryption ciphers, or more commonly, block encryption ciphers, such as the Data Encryption Standard [DES77], and triple DES variants, in conjunction with block chaining and checksum methods [DESM80]. 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 as defined in this section is chosen to minimize the effects of poorly chosen keys under certain types of cryptographic attacks. 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, as well as a framework for defining future mechanisms. The encodings, chaining, padding and other requirements for each

are described. Test vectors for several functions are given in appendix Z.

See also appendix Y for deprecated checksum algorithms defined in RFC 1510, which may still be in use by some implementations, but are not part of this version of the specification.

6.1. Concepts

Both encryption and checksum mechanism are defined in terms of profiles, detailed in later sections. Each specifies a collection of operations and attributes that must be defined for a mechanism. A Kerberos encryption or checksum mechanism is not complete if it does not specify all of these operations and attributes.

An encryption mechanism must provide for confidentiality and integrity of the original plaintext. (Integrity checking may be achieved by incorporating a checksum, if the encryption mode does not provide an integrity check itself.) It must also provide non-malleability [Bellare98, Dolev91]. Use of a random confounder prepended to the plaintext is recommended. It should not be possible to determine if two ciphertexts correspond to the same plaintext, without knowledge of the key.

A checksum mechanism[6.1] must provide proof of the integrity of the associated message, and must preserve the confidentiality of the message in case it is not sent in the clear. It should be infeasible to find two plaintexts which have the same checksum. It is NOT required that an eavesdropper be unable to determine if two checksums are for the same message; it is assumed that the messages themselves will be visible to any such eavesdropper.

Due to advances in cryptography, it is considered unwise by some cryptographers to use the same key for multiple purposes [@@reference??]. Since keys are used in performing a number of different functions in Kerberos, it is desirable to use different keys for each of these purposes, even though we start with a single long-term or session key.

We do this by enumerating the different uses of keys within Kerberos, and making the "usage number" an input to the encryption or checksum mechanisms. Later sections define simplified profile templates for encryption and checksum mechanisms that use a key derivation function applied to a CBC mode (or similar) cipher and a checksum or hash algorithm.

We distinguish the "base key" used in the EncryptedKey object from the "specific key" to be used for a particular instance of encryption or checksum operations. It is expected but not required that the specific key will be one or more separate keys derived from the original protocol key and the key usage number. The specific key is not explicitly referenced outside of section 6 in this document; when other sections mention encrypting or decrypting data with a given key, that key is the "base key", and the "specific key" generation and use is implicit, as is the use of a key usage number.

Key Usage Values

This is a list of key usage number definitions and reserved ranges, including values for all places keys are used in the Kerberos protocol and associated section numbers.

1. AS-REQ PA-ENC-TIMESTAMP padata timestamp, encrypted with the client key (section 5.4.1)
2. AS-REP Ticket and TGS-REP Ticket (includes TGS session key or application session key), encrypted with the service key (section 5.4.2)
3. AS-REP encrypted part (includes TGS session key or application session key), encrypted with the client key (section 5.4.2)
4. TGS-REQ KDC-REQ-BODY AuthorizationData, encrypted with the TGS session key (section 5.4.1)
5. TGS-REQ KDC-REQ-BODY AuthorizationData, encrypted with the TGS authenticator subkey (section 5.4.1)
6. TGS-REQ PA-TGS-REQ padata AP-REQ Authenticator cksum, keyed with the TGS session key (sections 5.3.2, 5.4.1)
7. TGS-REQ PA-TGS-REQ padata AP-REQ Authenticator (includes TGS authenticator subkey), encrypted with the TGS session key (section 5.3.2)
8. TGS-REP encrypted part (includes application session key), encrypted with the TGS session key (section 5.4.2)
9. TGS-REP encrypted part (includes application session key), encrypted with the TGS authenticator subkey (section 5.4.2)
10. AP-REQ Authenticator cksum, keyed with the application session key (section 5.3.2)
11. AP-REQ Authenticator (includes application authenticator subkey), encrypted with the application session key (section 5.3.2)
12. AP-REP encrypted part (includes application session subkey), encrypted with the application session key (section 5.5.2)
13. KRB-PRIV encrypted part, encrypted with a key chosen by the application (section 5.7.1)
14. KRB-CRED encrypted part, encrypted with a key chosen by the application (section 5.6.1)
15. KRB-SAFE cksum, keyed with a key chosen by the application (section 5.8.1)
18. KRB-ERROR checksum (e-cksum in section 5.9.1)
19. AD-KDCIssued checksum (ad-checksum in appendix B.4)
20. Checksum for Mandatory Ticket Extensions (appendix B.6)
21. Checksum in Authorization Data in Ticket Extensions (appendix B.7)

22-24. Reserved for use in GSSAPI mechanisms derived from RFC 1964. (raeburn/MIT)

25-511. Reserved for future use in Kerberos and related protocols.

512-1023. Reserved for uses internal to a Kerberos implementation. [6.2]

A few of these key usages need a little clarification. A service which receives an AP-REQ has no way to know if the enclosed Ticket was part of an AS-REP or TGS-REP. Therefore, key usage 2 must always be used for generating a Ticket, whether it is in response to an AS-REQ or TGS-REQ.

Key usage values between 1024 and 2047 (inclusive) are reserved for application use. Applications should use even values for encryption and odd values for checksums within this range.

There might exist other documents which define protocols in terms of the RFC1510 encryption types or checksum types. Such documents would not know about key usages. In order that these documents continue to be meaningful until they are updated, key usages 1024 and 1025 must be used to derive keys for encryption and checksums, respectively.[6.3] New protocols defined in terms of the Kerberos encryption and checksum types should use their own key usage values. Key usages are unsigned 32 bit integers; zero is not permitted. Usage numbers may be registered with IANA to avoid conflicts.

6.2. Encryption mechanism attributes

An encryption mechanism profile must define the following attributes and operations:

protocol key format
: This describes what OCTET STRING values represent valid keys. For encryption mechanisms that don't have perfectly dense key spaces, this will describe the representation used for encoding keys. It need not describe specific values that are not valid or desirable for use; such values should be avoid by all key generation routines.

specific key structure
: This is not a protocol format at all, but a description of the keying material derived from the chosen key and used to encrypt or decrypt data or compute or verify a checksum. It may be a single key, a set of keys, or a combination of the original key with additional data. The authors recommend using one or more keys derived from the original key via one-way functions.

required checksum mechanism
: This indicates a checksum mechanism that must be available when this encryption mechanism is used. Since Kerberos has no built in mechanism for negotiating checksum mechanisms, once an encryption mechanism has been decided upon, the corresponding checksum mechanism can simply be used.

key-generation seed length, K
: This is the length of the random bitstring needed to generate a key with the encryption scheme's random-to-key function (described below). This must be a fixed value so that various techniques for producing a

random bitstring of a given length may be used with key generation functions.

key generation functions

Keys must be generated in a number of cases, from different types of inputs. All function specifications must indicate how to generate keys in the proper wire format, and must avoid generation of "weak" keys if the cryptosystem has such. Entropy from each source should be preserved as much as possible. Many of the inputs, while unknown, may be at least partly predictable (e.g., a password string is likely to be entirely in the ASCII subset and of fairly short length in many environments; a semi-random string may include timestamps); the benefit of such predictability to an attacker must be minimized.

string-to-key (UTF8String, UTF8String, integer)->(protocol-key)

This function generates a key from two UTF-8 strings and an integer. One of the strings is normally the principal's password, but is in general merely a secret string. The other string is a "salt" intended to produce different keys from the same password for different users or realms. The integer is known for historical reasons as the "salt type", but may be used for selection between multiple string-to-key algorithms. The salt string and type may be altered by preauth data from the KDC; in the default case, the salt string is simply a concatenation of the principal's realm and name components, with no separators, and the salt type is zero.

This should be a one-way function, so that compromising a user's key in one realm does not compromise the user's key in another realm, even if the same password (but a different salt string) is used.

random-to-key (bitstring[K])->(protocol-key)

This function generates a key from a random bit string of a specific size. It may be assumed that all the bits of the input string are equally random, even though the entropy present in the random source may be limited.

combine-keys (protocol-key, protocol-key)->(protocol-key)

This function takes two input keys and produces a new key. This function is not used in this RFC, but may be used by preauthentication methods or other applications to be defined later. This operation must be commutative; this requirement lets us specify "combine keys A and B" in other documents without worrying about ordering.

key-derivation (protocol-key, integer)->(specific-key)

In this function, the byte string input is based on the key usage values specified above; the usage values must be assumed to be known to an attacker. For cryptosystems with dense key spaces, this function should be the key derivation function outlined in section 6.1.

cipher state
initial cipher state (specific-key)->(state)
     This describes any initial state setup needed before encrypting arbitrary amounts of data with a given specific key; the specific key must be the only input needed for this initialization. For example, a block cipher used in CBC mode must specify an initial vector. (For security reasons, the key itself should not be used as the IVEC.) This data may be carried over from one encryption or decryption operation to another. Unless otherwise specified, however, each encryption or decryption operation in this RFC uses a freshly initialized state and is thus independent of all other encryptions and decryptions.

     This state should be treated as opaque in any uses outside of an encryption algorithm definition.
encrypt (specific-key, state, bytestring)->(state, bytestring)
     This function takes the specific key, cipher state, and plaintext as input, and generates ciphertext and a new cipher state as outputs. If the basic encryption algorithm itself does not provide for integrity protection (as DES in CBC mode does not do), then some form of MAC or checksum must be included that can be verified by the receiver. Some random factor such as a confounder should be included so that an observer cannot know if two messages contain the same plaintext, even if the cipher state and specific keys are the same. The exact length of the plaintext need not be encoded, but if it is not and if padding is required, the padding must be added at the end of the string so that the decrypted version may be parsed from the beginning.

     The output will be used as the OCTET STRING in the EncryptedData type, defined in section 5.1.

decrypt (specific-key, state, bytestring)->(state, bytestring)
     This function takes the specific key, cipher state, and ciphertext as inputs, and verifies the integrity of the supplied ciphertext. If the ciphertext's integrity is intact, this function produces the plaintext and a new cipher state as outputs; otherwise, an error indication must be returned, and the data discarded.

     This function's byte string input is the OCTET STRING from the EncryptedData type, defined in section 5.1.

     The result of the decryption may be longer than the original plaintext, if the encryption mode requires padding to a multiple of a block size. If this is the case, any extra padding will be after the decoded plaintext. An application protocol which needs to know the exact length of the message must encode a length or recognizable "end of message" marker within the plaintext.

These operations and attributes are all that should be required to support

Kerberos and various proposed preauthentication schemes.

The reader is reminded that cryptography is a complex and growing field, and proof of the security of an algorithm is often difficult, even for those with extensive training in the field. Merely making an algorithm complicated is more likely to make it hard to analyze than it is to make it secure. This should be kept in mind when defining functions for new cryptosystems; simple applications of existing, trusted algorithms are more likely to be secure than a complicated home-grown scheme.

6.3. Checksum mechanism attributes

A checksum mechanism profile must define the following attributes and operations:

associated encryption algorithm(s)
: This essentially indicates the type of encryption key this checksum mechanism can be used with. A single checksum mechanism may have more than one associated encryption algorithm if they share the same wire key format, string-to-key function, and key derivation function. (This combination means that, for example, a checksum type and password are adequate to get the specific key used to compute a checksum.)

get_mic function
: This function generates a MIC token for a given specific key (see section 6.2), and message (represented as an octet string), that may be used to verify the integrity of the associated message. This function is not required to return the same deterministic result on every use; it need only generate a token that the verify_mic routine can check.

  The output of this function will also dictate the size of the checksum. It must be a fixed size.

verify_mic function
: Given a specific key, message, and MIC, this function ascertains whether the message integrity has been compromised. For a deterministic get_mic routine, the corresponding verify_mic may simply generate another checksum and compare them.

The get_mic and verify_mic operations must be able to handle inputs of arbitrary length; if any padding is needed, the padding scheme must be specified as part of these functions.

These operations and attributes are all that should be required to support Kerberos and various proposed preauthentication schemes.

6.4. A simplified profile for CBC-mode ciphers using key derivation

The profile outlines in sections 6.2 and 6.3 describes a large number of operations that must be defined for encryption and checksum algorithms to be

used with Kerberos. We describe here a simpler profile from which both encryption and checksum mechanism definitions can be generated, filling in uses of key derivation in appropriate places, providing integrity protection, and defining multiple operations for the cryptosystem profile based on a smaller set of operations given in the simplified profile. Not all of the existing cryptosystems for Kerberos fit into this simplified profile, but we recommend that future cryptosystems use it or something based on it.

Not all of the operations in the complete profiles are defined through this mechanism; several must still be defined for each new algorithm pair.

A key derivation function [Horowitz]

Rather than define some scheme by which a "protocol key" is composed of a large number of encryption keys, we use keys derived from a base key to perform cryptographic operations. The base key must be used only for generating the derived keys, and this derivation must be non-invertible and entropy-preserving. Given these restrictions, compromise of one derived key does not compromise the other subkeys. Attack of the base key is limited, since it is only used for derivation, and is not exposed to any user data.

Since the derived key has as much entropy as the base keys (if the cryptosystem is good), password-derived keys have the full benefit of all the entropy in the password.

To generate a derived key from a base key, we generate a pseudorandom byte string, using an algorithm DR described below, and generate a key from that byte string using a function dependent on the encryption algorithm; the input length needed for that function, which is also dependent on the encryption algorithm, dictates the length of the string to be generated by the DR algorithm (the value "k" below).

```
Derived Key = DK(Base Key, Well-Known Constant)

DK(Key, Constant) = random-to-key(DR(Key, Constant))

DR(Key, Constant) = k-truncate(E(Key, Constant))
```

Here DR is the random-byte generation function described below, and DK is the key-derivation function produced from it. In this construction, E(Key, Plaintext) is a block cipher, Constant is a well-known constant determined by the specific usage of this function, and k-truncate truncates its argument by taking the first k bits. Here, k is the key generation seed length needed for the encryption system.

The output of the DR function is a string of bits; the actual key is produced by applying the cryptosystem's random-to-key operation on this bitstring.

If the output of E is shorter than k bits, then some entropy in the key will be lost. If the Constant is smaller than the block size of E, then it must

be padded so it may be encrypted.

In either of these situations, a variation of the above construction is used, where the folded Constant is encrypted, and the resulting output is fed back into the encryption as necessary (the | indicates concatentation):


```
K1 = E(Key, n-fold(Constant))
K2 = E(Key, K1)
K3 = E(Key, K2)
K4 = ...

DR(Key, Constant) = k-truncate(K1 | K2 | K3 | K4 ...)
```

n-fold is an algorithm which takes m input bits and ``stretches'' them to form n output bits with equal contribution from each input bit to the output, as described in [Blumenthal96]:

> We first define a primitive called n-folding, which takes a variable-length input block and produces a fixed-length output sequence. The intent is to give each input bit approximately equal weight in determining the value of each output bit. Note that whenever we need to treat a string of bytes as a number, the assumed representation is Big-Endian -- Most Significant Byte first.
>
> To n-fold a number X, replicate the input value to a length that is the least common multiple of n and the length of X. Before each repetition, the input is rotated to the right by 13 bit positions. The successive n-bit chunks are added together using 1's-complement addition (that is, with end-around carry) to yield a n-bit result....

Test vectors for n-fold are supplied in Appendix Z. [6.4]

In this document, n-fold is always used to produce n bits of output, where n is the block size of E.

The size of the Constant must not be larger than the block size of E, because reducing the length of the Constant by n-folding can cause collisions.

If the size of the Constant is smaller than the block size of E, then the Constant must be n-folded to the block size of E. This string is used as input to E. If the block size of E is less than the key size, then the output from E is taken as input to a second invocation of E. This process is repeated until the number of bits accumulated is greater than or equal to the key size of E. When enough bits have been computed, the first k are taken as the derived key.

Since the derived key is the result of one or more encryptions in the base key, deriving the base key from the derived key is equivalent to determining the key from a very small number of plaintext/ciphertext pairs. Thus, this construction is as strong as the cryptosystem itself.

6.4.1. Simplified profile parameters

These are the operations and attributes that must be defined:

protocol key format
string-to-key function
key-generation seed length, k
random-to-key function
: As above for the normal encryption mechanism profile.

unkeyed hash algorithm, H
: This should be a collision-resistant hash algorithm such as SHA-1, suitable for use in an HMAC. It must support inputs of arbitrary length.

encryption block size, n
encryption/decryption functions, E and D
: These are basic encryption and decryption functions for messages of sizes that are multiples of the block size. No integrity checking or confounder should be included here. They take as input the IV or similar data, a protocol-format key, and a byte string, returning a new IV and byte string.

  The encryption function is not required to use CBC mode, but is assumed to be using something with similar properties. In particular, prepending a one-block confounder to the plaintext should alter the entire ciphertext (comparable to choosing and including a random initial vector for CBC mode).

While there are still a number of properties to specify, they are fewer and simpler than in the full profile.

6.4.2. Cryptosystem profile based on simplified profile

protocol key format	As given.
specific key structure	Three protocol-format keys: { Kc, Ke, Ki }.
key-generation seed length	As given.
required checksum mechanism	The checksum mechanism defined by the simplified checksum profile given later.
cipher state	Initial vector, initialized to all zero.
encryption	The ciphertext output is the concatenation of the output of

function — the basic encryption function E and an HMAC using the specified hash function H, both applied to the padded plaintext with a confounder:

```
ciphertext =    E(Ke, confounder | plaintext | padding) |
             HMAC(Ki, confounder | plaintext | padding)
HMAC(K,M) = H(K | H(K | M))
```

One block of random confounder data is prepended to the plaintext, and padding added to the end to bring the length to a multiple of the encryption algorithm's block size. The initial vector for encryption is supplied by the cipher state, and the last block of the output of E is the new IVEC for the new cipher state.

decryption function — Decryption is performed by extracting the encrypted portion of the ciphertext, decrypting using key Ke from the specific key, and verifying the HMAC. If the HMAC is incorrect, an error must be reported. Otherwise, the confounder and padding are discarded and the remaining plaintext returned. As with encryption, the cipher state input indicates the IVEC to use, and the last block of the encrypted portion of the ciphertext is put into the new cipher state to be used as the next IVEC.

key generation functions:

string-to-key function — As given.

random-to-key function — As given.

combine-keys function — @@ Needs to be specified. How about:

```
combine-keys(K1,K2)
    /* First, protect original keys against exposure
       with DR.  */
    R1 = DR(K1, n-fold(K2))    /* length k */
    R2 = DR(K2, n-fold(K1))    /* length k */
    /* Using k-fold on length 2k means just add with
       wrap-around carry.  */
    rnd = k-fold(R1 | R2)
    tkey = random-to-key(rnd)
    key = DK(tkey, CombineConstant)
```

@@ This should be commutative, which keeps the specifications simpler, and I think should protect each key's contribution from exposure of the other key. Or should we just go with an XOR combination of the random bytes? (But that works poorly should the two keys happen to be the same due to sloppy or weird protocol specs, or

parallel construction of contributed keys.) Do we need the two DK invocations at the start? Would one at the end be better? Do we need either?

Here CombineConstant is the byte string {0x63 0x6f 0x6d 0x62 0x69 0x6e 0x65} corresponding to the ASCII encoding of the string "combine".

@@ Need a cryptographer to review this. Asked Uri Blumenthal, he said he'd look it over when he has time.

key-derivation function

Three keys are generated, using the DK function described above, and the key usage number, represented as a 32-bit integer in big-endian byte order. One is used for generating checksums only; the other two are used for encrypting and integrity protection for ciphertext. These keys are generated as follows (with "|" indicating concatenation):

```
Kc = DK(base-key, usage|0x99));
Ke = DK(base-key, usage|0xAA);
Ki = DK(base-key, usage|0x55);
```

6.4.3. Checksum profiles based on simplified profile

When an encryption system is defined using the simplified profile given in section 6.4.1, a checksum algorithm may be defined for it as follows:

associated cryptosystem	as defined above
get_mic	HMAC(Kc, message)
verify_mic	get_mic and compare

The HMAC function and key Kc are as described in section 6.4.2.

6.5. Profiles for Kerberos encryption systems

These are the currently defined encryption systems for Kerberos. The astute reader will notice that some of them do not fulfill all of the requirements outlined above. These weaker encryption systems are defined for backwards compatibility; newer implementations should attempt to make use of the stronger encryption systems when possible.

The full list of encryption type number assignments is given in section 8.3.

6.5.1. 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.

This encryption system should not be used for protection of data. It exists primarily to associate with the rsa-md5 checksum type, but may also be useful for testing protocol implementations.

protocol key format	zero-length bit string
specific key structure	empty
required checksum mechanism	rsa-md5
key-generation seed length	0
cipher state	none
initial cipher state	none
encryption function	identity
decryption function	identity, no integrity check
key generation functions:	
string-to-key	empty string
random-to-key	empty string
combine-keys	empty string
key-derivation	empty string

The null encryption algorithm is assigned the etype value zero (0).

6.5.2. des-cbc-md5

The des-cbc-md5 encryption mode encrypts information under the Data Encryption Standard [DES77] using the cipher block chaining mode [DESM80]. An MD5 checksum (described in [MD5-92]) 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 blocks of 8 octets 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.

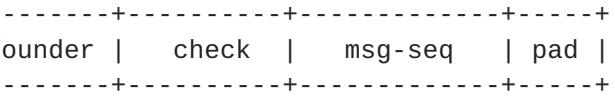
The DES specifications identify some 'weak' and 'semi-weak' keys; those keys shall not be used for encrypting messages for use in Kerberos. Additionally, because of the way that keys are derived for the encryption of checksums, keys shall not be used that yield 'weak' or 'semi-weak' keys when eXclusive-ORed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

Encryption data format

The format for the data to be encrypted includes a one-block confounder, a checksum, the encoded plaintext, and any necessary padding, as described in the following diagram. 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.



One generates a random confounder of one block, 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.

String to key transformation

To generate a DES key from a UTF-8 text string (password), a "salt" is concatenated to the text string, and 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. Before eXclusive-ORing a block, every byte is shifted one bit to the left to leave the lowest bit zero. The key is the "corrected" by correcting the parity on the key, and if the key matches a 'weak' or 'semi-weak' key as described in the DES specification, it is eXclusive-ORed with the constant 00000000000000F0. This key is then used to generate a DES CBC checksum on the initial string (with the salt appended). The result of the CBC checksum is the "corrected" as described above to form the result which is return as the key.

Pseudocode follows:

```
key_correction(key) {
     fixparity(key);
     if (is_weak_key_key(key))
          key = key XOR 0xF0;
     return(key);
}

mit_des_string_to_key(string,salt) {
```

```
          odd = 1;
          s = string + salt;
          tempkey = NULL;
          pad(s); /* with nulls to 8 byte boundary */
          for (8byteblock in s) {
               if (odd == 0)  {
                   odd = 1;
                   reverse(8byteblock)
               }
               else odd = 0;
               left shift every byte in 8byteblock one bit;
               tempkey = tempkey XOR 8byteblock;
          }
          tempkey = key_correction(tempkey);
          key = key_correction(DES-CBC-check(s,tempkey));
          return(key);
     }
```



```
     des_string_to_key(type,string,salt) {
          if (type == 0)
               mit_des_string_to_key(string,salt);
          else if (type == 1)
               afs_des_string_to_key(string,salt);
     }
```

The AFS string-to-key algorithm is not defined here, but salt type value one (1) is reserved for that use.

The encryption system parameters for des-cbc-md5 are:

protocol key format	8 bytes, parity in low bit of each
specific key structure	copy of original key
required checksum mechanism	rsa-md5-des
key-generation seed length	8 bytes
cipher state	8 bytes (CBC initial vector)
initial cipher state	all-zero
encryption function	des-cbc(confounder | checksum | msg | pad)
	with checksum computed as described above
decryption function	decrypt encrypted text and verify checksum
key generation functions:	
string-to-key	des_string_to_key
random-to-key	copy input, then fix parity bits (discards low bit of each input byte)
combine-keys	bitwise XOR, then fix parity bits
key-derivation	identity

The des-cbc-md5 encryption type is assigned the etype value three (3).

6.5.3. des-cbc-md4

The des-cbc-md4 encryption mode encrypts information under the Data Encryption Standard [DES77] using the cipher block chaining mode [DESM80]. An MD4 checksum (described in [MD492]) 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.

protocol key format	8 bytes, parity in low bit of each
specific key structure	copy of original key
required checksum mechanism	rsa-md4-des
key-generation seed length	8 bytes
cipher state	8 bytes (CBC initial vector)
initial cipher state	all-zero
encryption function	des-cbc(confounder | checksum | msg | pad)
	with checksum computed as described above
decryption function	decrypt encrypted text and verify checksum
key generation functions:	
string-to-key	des_string_to_key
random-to-key	copy input, then fix parity bits
combine-keys	bitwise XOR, then fix parity bits
key-derivation	identity

The des-cbc-md4 encryption algorithm is assigned the etype value two (2).

6.5.4. des-cbc-crc

The des-cbc-crc encryption mode encrypts information under the Data Encryption Standard [DES77] using the cipher block chaining mode [DESM80]. A 4-octet CRC-32 checksum (described in ISO 3309 [ISO3309]) is computed over 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. Unless otherwise specified, the key should be used as the initialization vector, unlike for the other Kerberos DES encryption schemes. The other 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 collision-proof, an attacker could use a probabilistic chosen-plaintext attack to generate a valid message even if a confounder is used [SG92]. 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).

protocol key format	8 bytes, parity in low bit of each
specific key structure	copy of original key
required checksum mechanism	rsa-md5-des
key-generation seed length	8 bytes
cipher state	8 bytes (CBC initial vector)
initial cipher state	copy of original key
encryption function	des-cbc(confounder | checksum | msg | pad)
	with checksum computed as described above
decryption function	decrypt encrypted text and verify checksum
key generation functions:	
string-to-key	des_string_to_key
random-to-key	copy input, then fix parity bits
combine-keys	bitwise XOR, then fix parity bits
key-derivation	identity

The des-cbc-crc encryption algorithm is assigned the etype value one (1).

6.5.5. des3-cbc-hmac-sha1-kd

This encryption type is based on the Triple DES cryptosystem in Outer-CBC mode, and the HMAC-SHA1 [Krawczyk96] message authentication algorithm.

A Triple DES key is the concatenation of three DES keys as described above for des-cbc-md5. A Triple DES key is generated from random data by creating three DES keys from separate sequences of random data.

EncryptedData using this type must be generated as described in section 6.4.2. **If the length of the input data is not a multiple of the block size,** zero octets must be used to pad the plaintext to the next eight-octet boundary. The counfounder must be eight random octets (one block).

The simplified profile for Triple DES, with key derivation as defined in section 6.4, is as follows:

protocol key format	24 bytes, parity in low bit of each
key-generation seed length	21 bytes
hash function	SHA-1
block size	8 bytes
encryption, decryption functions	triple-DES EDE in outer CBC mode
key generation functions:	
random-to-key	see below
string-to-key	DES3string-to-key (see below)

The des3-cbc-hmac-sha1-kd encryption type is assigned the value sixteen (16).

Triple DES Key Production (random-to-key, string-to-key)

The 168 bits of random key data are converted to a protocol key value as follows. First, the 168 bits are divided into three groups of 56 bits, which are expanded individually into 64 bits as follows:

```
 1  2  3  4  5  6  7  p
 9 10 11 12 13 14 15  p
17 18 19 20 21 22 23  p
25 26 27 28 29 30 31  p
33 34 35 36 37 38 39  p
41 42 43 44 45 46 47  p
49 50 51 52 53 54 55  p
56 48 40 32 24 16  8  p
```

The "p" bits are parity bits computed over the data bits. The output of the three expansions are concatenated to form the protocol key value.

When the HMAC-SHA1 of a string is computed, the key is used in the protocol key form.

The string-to-key function is used to tranform UTF-8 passwords into DES3 keys. The DES3 string-to-key function relies on the "N-fold" algorithm and DK function, described in section 6.4.

The n-fold algorithm is applied to the password string concatenated with a salt value. For 3-key triple DES, the operation will involve a 168-fold of the input password string, to generate an intermediate key, from which the user's long-term key will be derived with the DK function. The DES3 string-to-key function is shown here in pseudocode:

```
DES3string-to-key(passwordString, salt, key)

    s = passwordString + salt
    tmpKey = random-to-key(168-fold(s))
    key = DK (tmpKey, KerberosConstant)
```

No weak-key checking is performed. The KerberosConstant value is the byte string {0x6b 0x65 0x72 0x62 0x65 0x72 0x6f 0x73}. These values correspond to the ASCII encoding for the string "kerberos".

6.6. Profiles for Kerberos checksums

These are the checksum types currently defined for Kerberos. The full list of checksum type number assignments is given in section 8.3.

6.6.1. RSA MD4 Cryptographic Checksum Using DES (rsa-md4-des)

The RSA-MD4-DES checksum calculates a keyed collision-proof checksum by

prepending an 8 octet confounder before the text, applying the RSA MD4 checksum algorithm [MD4-92], 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[39]. The initialization vector should be zero. The resulting checksum is 24 octets long. This checksum is tamper-proof and believed to be collision-proof.

The DES specifications identify some weak keys' and 'semi-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)  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
 associated cryptosystem des-cbc-md5, des-cbc-md4, des-cbc-crc
 get_mic                 des-cbc(key XOR F0F0F0F0F0F0F0F0,
                                 confounder | rsa-md4(confounder | msg))

 verify_mic              decrypt and verify rsa-md4 checksum
```

The rsa-md4-des checksum algorithm is assigned a checksum type number of three (3).

6.6.2. The RSA MD5 Checksum (rsa-md5)

The RSA-MD5 checksum calculates a checksum using the RSA MD5 algorithm [MD5-92]. 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. However, since it is unkeyed, it must be used with caution. Currently it is used by some implementations in places where the checksum itself is part of a larger message that will be encrypted. Its use is not recommended.

```
 associated cryptosystem null
 get_mic                 rsa-md5(msg)
 verify_mic              get_mic and compare
```

The rsa-md5 checksum algorithm is assigned a checksum type number of seven (7).

6.6.3. RSA MD5 Cryptographic Checksum Using DES (rsa-md5-des)

The RSA-MD5-DES checksum calculates a keyed collision-proof 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 hexadecimal constant F0F0F0F0F0F0F0F0. The initialization vector should be zero. The resulting checksum is 24 octets long. This checksum is tamper-proof and believed to be

collision-proof.

The DES specifications identify some 'weak keys' and 'semi-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)  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
 associated cryptosystem des-cbc-md5, des-cbc-md4, des-cbc-crc
 get_mic                 des-cbc(key XOR F0F0F0F0F0F0F0F0,
                                 confounder | rsa-md5(confounder | msg))

 verify_mic              decrypt and verify rsa-md5 checksum
```

The rsa-md5-des checksum algorithm is assigned a checksum type number of eight (8).

6.6.4. The HMAC-SHA1-DES3-KD Checksum (hmac-sha1-des3-kd)

This checksum type is defined as outlined in section 6.3 above, using the des3-hmac-sha1-kd encryption algorithm parameters. The checksum is thus a SHA-1 HMAC using the computed key Kc over the message to be protected.

The hmac-sha1-des3-kd checksum algorithm is assigned a checksum type number of twelve (12).

6.7. Use of Kerberos encryption outside this specification

Several Kerberos-based application protocols and preauthentication systems have been designed and deployed that perform encryption and message integrity checks in various ways. While in some cases there may be good reason for specifying these protocols in terms of specific encryption or checksum algorithms, we anticipate that in many cases this will not be true, and more generic approaches independent of particular algorithms will be desirable. Rather than having each protocol designer reinvent schemes for protecting data, using multiple keys, etc, we have attempted to present in this section a general framework that should be sufficient not only for the Kerberos protocol itself but also for many preauthentication systems and application protocols, while trying to avoid some of the assumptions that can work their way into such protocol designs.[6.5] Such assumptions, while they may hold for any given set of encryption and checksum algorithms, may not be true of the next algorithms to be defined, leaving the application protocol unable to make use of those algorithms without updates to its specification.

The Kerberos protocol uses only the attributes and operations described in

sections 6.2 and 6.3. Preauthentication systems and application protocols making use of Kerberos are encouraged to use them as well.

While we don't recommend it, undoubtedly some application protocols will continue to use the key data directly, even if only in some of the currently existing protocol specifications. An implementation intended to support general Kerberos applications may therefore need to make the key data available, as well as the attributes and operations described in sections **6.2 and 6.3. [6.6]**

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.

Kerberos realm names are case sensitive. Realm names that differ only in the case of the characters are not equivalent. There are presently four styles of realm names: domain, X500, other, and reserved. Examples of each style follow:

```
     domain:   ATHENA.MIT.EDU (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 (/). Though domain names themselves are case insensitive, in order for realms to match, the case must match as well. When establishing a new realm name based on an internet domain name it is recommended by convention that the characters be converted to upper case.

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. Note that the slash separator is consistent with Kerberos implementations based on RFC1510, but it is different from the separator recommended in RFC2253.

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 exist 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). The following name types are defined:

name-type	value	meaning
NT-UNKNOWN	0	Name type not known
NT-PRINCIPAL	1	General principal name (e.g. username, or DCE principal)
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 slash-separated host name components
NT-UID	5	Unique ID
NT-X500-PRINCIPAL	6	Encoded X.509 Distingished name [RFC 1779]
NT-SMTP-NAME	7	Name in form of SMTP email name (e.g. user@foo.com)

NT-ENTERPRISE 10 Enterprise name - may be mapped to principal name

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 service whose name 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 SRV-XHST 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 NT-X500-PRINCIPAL should be used when a name from an X.509 certificate is translated into a Kerberos name. The encoding of the X.509 name as a Kerberos principal shall conform to the encoding rules specified in RFC 2253.

A name type of SMTP allows a name to be of a form that resembles a SMTP email name. This name, including an "@" and a domain name, is used as the one component of the principal name.

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. If specified by the application protocol for services such as telnet and the Berkeley R commands which run with system privileges, the first component may be the string 'host' instead of a service specific identifier. When a host has an official name and one or more aliases and the official name can be reliably determined, the official name of the host should be used when constructing the name of the server principal.

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 with symbolic names AF_xxx (where xxx is an abbreviation of the address family name).

Internet (IPv4) Addresses

Internet (IPv4) addresses are 32-bit (4-octet) quantities, encoded in MSB order. The IPv4 loopback address should not appear in a Kerberos packet. The type of IPv4 addresses is two (2).

Internet (IPv6) Addresses [Westerlund]

IPv6 addresses are 128-bit (16-octet) quantities, encoded in MSB order. The type of IPv6 addresses is twenty-four (24). [RFC2373]. The following addresses (see [RFC1884]) MUST not appear in any Kerberos packet:

* the Unspecified Address
* the Loopback Address
* Link-Local addresses

IPv4-mapped IPv6 addresses MUST be represented as addresses of type 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).

Netbios addresses

Netbios addresses are 16-octet addresses typically composed of 1 to 15 characters, trailing blank (ascii char 20) filled, with a 16th octet of 0x0. The type of Netbios addresses is 20 (0x14).

8.2. KDC messages

8.2.1. UDP/IP transport

When contacting a Kerberos server (KDC) for a KRB_KDC_REQ request using UDP 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. Kerberos servers supporting IP transport must accept UDP requests on port 88 (decimal). The response to a request made through UDP/IP transport must also use UDP/IP transport.

8.2.2. TCP/IP transport [Westerlund,Danielsson]

Kerberos servers (KDC's) should accept TCP requests on port 88 (decimal) and clients should support the sending of TCP requests on port 88 (decimal). When the KRB_KDC_REQ message is sent to the KDC over a TCP stream, a new connection will be established for each authentication exchange (request and response). The KRB_KDC_REP or KRB_ERROR message will be returned to the client on the same TCP stream that was established for the request. The response to a request made through TCP/IP transport must also use TCP/IP transport. Implementors should note that some extensions to the Kerberos protocol will not work if any implementation not supporting the TCP transport is involved (client or KDC). Implementors are strongly urged to

support the TCP transport on both the client and server and are advised that

the current notation of "should" support will likely change in the future to must support. The KDC may close the TCP stream after sending a response, but may leave the stream open if it expects a followup - in which case it may close the stream at any time if resource constraints or other factors make it desirable to do so. Care must be taken in managing TCP/IP connections with the KDC to prevent denial of service attacks based on the number of TCP/IP connections with the KDC that remain open. If multiple exchanges with the KDC are needed for certain forms of preauthentication, multiple TCP connections may be required. A client may close the stream after receiving response, and should close the stream if it does not expect to send followup messages. The client must be prepared to have the stream closed by the KDC at anytime, in which case it must simply connect again when it is ready to send subsequent messages.

The first four octets of the TCP stream used to transmit the request request will encode in network byte order the length of the request (KRB_KDC_REQ), and the length will be followed by the request itself. The response will similarly be preceded by a 4 octet encoding in network byte order of the length of the KRB_KDC_REP or the KRB_ERROR message and will be followed by the KRB_KDC_REP or the KRB_ERROR response. If the sign bit is set on the integer represented by the first 4 octets, then the next 4 octets will be read, extending the length of the field by another 4 octets (less the sign bit of the additional four octets which is reserved for future expansion and which at present must be zero).

8.2.3. OSI transport

During authentication of an OSI client to an 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(6),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.

8.2.4. 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-SRV-INST, 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 define their meanings. Ranges are specified in the "specification" section that limit the values of constants for which values are defined here. This allows implementations to make assumptions about the maximum values that will be received for these constants. Implementation receiving values outside the range specified in the "specification" section may reject the request, but they must recover cleanly.

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
[reserved]	4			
des3-cbc-md5	5	8	0	8
[reserved]	6			
des3-cbc-sha1	7	8	0	8
dsaWithSHA1-CmsOID	9			(pkinit)
md5WithRSAEncryption-CmsOID	10			(pkinit)
sha1WithRSAEncryption-CmsOID	11			(pkinit)
rc2CBC-EnvOID	12			(pkinit)
rsaEncryption-EnvOID	13		(pkinit from PKCS#1 v1.5)	
rsaES-OAEP-ENV-OID	14		(pkinit from PKCS#1 v2.0)	
des-ede3-cbc-Env-OID	15			(pkinit)
des3-cbc-sha1-kd	16			(Tom Yu)
rc4-hmac	23			(swift)
rc4-hmac-exp	24			(swift)
subkey-keynaterial	65			(opaque mhur)

[reserved] 0x8003

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 (drop rsa ?)
rsa-md5	7	16 (drop rsa ?)
rsa-md5-des	8	24 (drop rsa ?)
rsa-md5-des3	9	24 (drop rsa ?)
hmac-sha1-des3-kd	12	20
hmac-sha1-des3	13	20
sha1 (unkeyed)	14	20

padata and data types	padata-type value	comment
PA-TGS-REQ	1	
PA-ENC-TIMESTAMP	2	
PA-PW-SALT	3	
[reserved]	4	
PA-ENC-UNIX-TIME	5	(depricated)
PA-SANDIA-SECUREID	6	
PA-SESAME	7	
PA-OSF-DCE	8	
PA-CYBERSAFE-SECUREID	9	
PA-AFS3-SALT	10	
PA-ETYPE-INFO	11	
PA-SAM-CHALLENGE	12	(sam/otp)
PA-SAM-RESPONSE	13	(sam/otp)
PA-PK-AS-REQ	14	(pkinit)

PA-PK-AS-REP	15	(pkinit)
PA-USE-SPECIFIED-KVNO	20	
PA-SAM-REDIRECT	21	(sam/otp)
PA-GET-FROM-TYPED-DATA	22	(embedded in typed data)
TD-PADATA	22	(embeds padata)
PA-SAM-ETYPE-INFO	23	(sam/otp)
PA-ALT-PRINC	24	(crawdad@fnal.gov)
TD-PKINIT-CMS-CERTIFICATES	101	CertificateSet from CMS
TD-KRB-PRINCIPAL	102	PrincipalName (see Sec.5.9.1)
TD-KRB-REALM	103	Realm (see Sec.5.9.1)
TD-TRUSTED-CERTIFIERS	104	from PKINIT
TD-CERTIFICATE-INDEX	105	from PKINIT
TD-APP-DEFINED-ERROR	106	application specific (see Sec.5.9.1)
TD-REQ-NONCE	107	INTEGER (see Sec.5.9.1)
TD-REQ-SEQ	108	INTEGER (see Sec.5.9.1)
PA-PAC-REQUEST	128	(jbrezak@exchange.microsoft.com)

authorization data type	ad-type value
AD-IF-RELEVANT	1
AD-INTENDED-FOR-SERVER	2
AD-INTENDED-FOR-APPLICATION-CLASS	3
AD-KDC-ISSUED	4
AD-OR	5

AD-MANDATORY-TICKET-EXTENSIONS	6	
AD-IN-TICKET-EXTENSIONS	7	
reserved values	8-63	
OSF-DCE	64	
SESAME	65	
AD-OSF-DCE-PKI-CERTID	66	(hemsath@us.ibm.com)
AD-WIN2K-PAC	128	(jbrezak@exchange.microsoft.com)

Ticket Extension Types

TE-TYPE-NULL	0	Null ticket extension
TE-TYPE-EXTERNAL-ADATA	1	Integrity protected authorization data
[reserved]	2	TE-TYPE-PKCROSS-KDC (I have reservations)
TE-TYPE-PKCROSS-CLIENT	3	PKCROSS cross realm key ticket
TE-TYPE-CYBERSAFE-EXT	4	Assigned to CyberSafe Corp
[reserved]	5	TE-TYPE-DEST-HOST (I have reservations)

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 (Will be updated to match section 5)

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
KRB_NT_X500_PRINCIPAL 6  Encoded X.509 Distingished name [RFC 2253]

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-authenticationrequired [40]
KDC_ERR_SERVER_NOMATCH         26   Requested server and ticket don't match
KDC_ERR_MUST_USE_USER2USER     27   Server principal valid for user2user only
KDC_ERR_PATH_NOT_ACCPETED      28   KDC Policy rejects transited path
KDC_ERR_SVC_UNAVAILABLE        29   A service is not available
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_AP_PATH_NOT_ACCEPTED	51	Policy rejects transited path
KRB_ERR_RESPONSE_TOO_BIG	52	Response too big for UDP, retry with TCP
KRB_ERR_GENERIC	60	Generic error (description in e-text)
KRB_ERR_FIELD_TOOLONG	61	Field is too long for this implementation
KDC_ERROR_CLIENT_NOT_TRUSTED	62	(pkinit)
KDC_ERROR_KDC_NOT_TRUSTED	63	(pkinit)
KDC_ERROR_INVALID_SIG	64	(pkinit)
KDC_ERR_KEY_TOO_WEAK	65	(pkinit)
KDC_ERR_CERTIFICATE_MISMATCH	66	(pkinit)
KRB_AP_ERR_NO_TGT	67	(user-to-user)
KDC_ERR_WRONG_REALM	68	(user-to-user)
KRB_AP_ERR_USER_TO_USER_REQUIRED	69	(user-to-user)
KDC_ERR_CANT_VERIFY_CERTIFICATE	70	(pkinit)
KDC_ERR_INVALID_CERTIFICATE	71	(pkinit)
KDC_ERR_REVOKED_CERTIFICATE	72	(pkinit)
KDC_ERR_REVOCATION_STATUS_UNKNOWN	73	(pkinit)
KDC_ERR_REVOCATION_STATUS_UNAVAILABLE	74	(pkinit)
KDC_ERR_CLIENT_NAME_MISMATCH	75	(pkinit)
KDC_ERR_KDC_NAME_MISMATCH	76	(pkinit)

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 2

This section defines the second specification of these options. Implementations which are configured in this way can be said to support

Kerberos Version 5 Specification 2 (5.1). Specification 1 (deprecated) may be found in RFC1510.

Transport

TCP/IP and UDP/IP transport must be supported by KDCs claiming conformance to specification 2. Kerberos clients claiming conformance to specification 2 must support UDP/IP transport for messages with the KDC and should support TCP/IP transport.

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: This list is to be determined and should correspond to section 6.

Encryption: DES-CBC-MD5, DES3-CBC-SHA1-KD, RIJNDAEL(decide identifier)
Checksums: CRC-32, DES-MAC, DES-MAC-K, DES-MD5, HMAC-SHA1-DES3-KD

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 3.3.3.2) 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-ENC-TIMESTAMP as an acceptable method, the client should retry the initial request using the PA-ENC-TIMESTAMP preauthentication method. Servers need not support the PA-ENC-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 through 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.

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-TKT-IN-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.

Constant ranges

All protocol constants are constrained to 32 bit (signed) values unless further constrained by the protocol definition. This limit is provided to allow implementations to make assumptions about the maximum values that will be received for these constants. Implementation receiving values outside this range may reject the request, but they must recover cleanly.

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. IANA considerations

An appendix will be created with the tables that IANA will need to start maintaining.

* cryptosystem registration
* usage number registration

11. ACKNOWLEDGEMENTS

T.B.S.

12. REFERENCES

[Blumenthal96]
Blumenthal, U., "A Better Key Schedule for DES-Like Ciphers", Proceedings of PRAGOCRYPT '96, 1996.

[Bellare98]
Bellare, M., Desai, A., Pointcheval, D., Rogaway, P., "Relations Among Notions of Security for Public-Key Encryption Schemes". Extended abstract published in Advances in Cryptology- Crypto 98 Proceedings, Lecture Notes in Computer Science Vol. 1462, H. Krawcyzk ed., Springer-Verlag, 1998.

[DES77]
National Bureau of Standards, U.S. Department of Commerce, "Data Encryption Standard," Federal Information Processing Standards Publication 46, Washington, DC (1977).

[DESM80]
National Bureau of Standards, U.S. Department of Com- merce, "DES Modes of Operation," Federal Information Processing Standards Publication 81, Springfield, VA (December 1980).

[Dolev91]
Dolev, D., Dwork, C., Naor, M., "Non-malleable cryptography", Proceedings of the 23rd Annual Symposium on Theory of Computing, ACM, 1991.

[DS81]
Dorothy E. Denning and Giovanni Maria Sacco, "Time- stamps in Key Distribution Protocols," Communications of the ACM, Vol. 24(8), pp. 533-536 (August 1981).

[DS90]
Don Davis and Ralph Swick, "Workstation Services and Kerberos Authentication at Project Athena," Technical Memorandum TM-424, MIT

Laboratory for Computer Science (February 1990).
[Horowitz96]
Horowitz, M., "Key Derivation for Authentication, Integrity, and Privacy", draft-horowitz-key-derivation-02.txt, August 1998.
[HorowitzB96]
Horowitz, M., "Key Derivation for Kerberos V5", draft-

horowitz-kerb-key-derivation-01.txt, September 1998.
[IS3309]
International Organization for Standardization, "ISO Information Processing Systems - Data Communication - High-Level Data Link Control Procedure - Frame Struc- ture," IS 3309 (October 1984). 3rd Edition.
[KBC96]
H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed- Hashing for Message Authentication," Working Draft draft-ietf-ipsec-hmac-md5-01.txt, (August 1996).
[KNT92]
John T. Kohl, B. Clifford Neuman, and Theodore Y. Ts'o, "The Evolution of the Kerberos Authentication Service," in an IEEE Computer Society Text soon to be published (June 1992).
[Krawczyk96]
Krawczyk, H., Bellare, and M., Canetti, R., "HMAC: Keyed-Hashing for Message Authentication", draft-ietf-ipsec-hmac- md5-01.txt, August, 1996.
[LGDSR87]
P. J. Levine, M. R. Gretzinger, J. M. Diaz, W. E. Som- merfeld, and K. Raeburn, Section E.1: Service Manage- ment System, M.I.T. Project Athena, Cambridge, Mas- sachusetts (1987).
[MD4-92]
R. Rivest, "The MD4 Message Digest Algorithm," RFC 1320, MIT Laboratory for Computer Science (April 1992).
[MD5-92]
R. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, MIT Laboratory for Computer Science (April 1992).
[MNSS87]
S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, Section E.2.1: Kerberos Authentication and Authorization System, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).
[Neu93]
B. Clifford Neuman, "Proxy-Based Authorization and Accounting for Distributed Systems," in Proceedings of the 13th International Conference on Distributed Com- puting Systems, Pittsburgh, PA (May, 1993).
[NS78]
Roger M. Needham and Michael D. Schroeder, "Using Encryption for Authentication in Large Networks of Com- puters," Communications of the ACM, Vol. 21(12), pp. 993-999 (December, 1978).
[NT94]
B. Clifford Neuman and Theodore Y. Ts'o, "An Authenti- cation Service

for Computer Networks," IEEE Communica- tions Magazine, Vol. 32(9), pp. 33-38 (September 1994).

[Pat92].
J. Pato, Using Pre-Authentication to Avoid Password Guessing Attacks, Open Software Foundation DCE Request for Comments 26 (December 1992).

[SG92]
Stuart G. Stubblebine and Virgil D. Gligor, "On Message Integrity in Cryptographic Protocols," in Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, California (May 1992).

[SNS88]
J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Ker- beros: An Authentication Service for Open Network Sys- tems," pp. 191-202 in Usenix Conference Proceedings, Dallas, Texas (February, 1988).

[X509-88]
CCITT, Recommendation X.509: The Directory Authentica- tion Framework, December 1988.

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. [we are waiting on verification code from Microsoft for possible replacement]

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.MAY-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.POSTDATED;
           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(req.sname)) then
                        server := best_intermediate_tgs(req.sname);
                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.ALLOW-POSTDATE is set) then
        if (tgt.flags.MAY-POSTDATE is reset)
                error_out(KDC_ERR_BADOPTION);
        endif
        set new_tkt.flags.MAY-POSTDATE;
endif
if (req.kdc-options.POSTDATED is set) then
        if (tgt.flags.MAY-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.endttime - 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)
                /* Don't check tranited field if TGT for foreign realm,=20
                 * or requested not to check */
                if (is_not_foreign_tgt_name(new_tkt.server)=20
                   && req.kdc-options.DISABLE-TRANSITED-CHECK not set) then
                        /* Check it, so end-server does not have to=20
                         * but don't fail, end-server may still accept it */
                        if (check_transited_field(new_tkt.transited) == OK)
                              set =
new_tkt.flags.TRANSITED-POLICY-CHECKED;
                        endif
                endif
        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);

=09
```

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
        if (decr_ticket.transited) then
            /* caller may ignore the TRANSITED-POLICY-CHECKED and do
             * check anyway */
            if (decr_ticket.flags.TRANSITED-POLICY-CHECKED not set) then
                 if (check_transited_field(decr_ticket.transited) then
```



```
                     error_out(KDC_AP_PATH_NOT_ACCPETED);
                 endif
            endif
        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
```


B. Definition of common authorization data elements

This appendix contains the definitions of common authorization data elements. These common authorization data elements are recursivly defined, meaning the ad-data for these types will itself contain a sequence of authorization data whose interpretation is affected by the encapsulating element. Depending on the meaning of the encapsulating element, the encapsulated elements may be ignored, might be interpreted as issued directly by the KDC, or they might be stored in a separate plaintext part of the ticket. The types of the encapsulating elements are specified as part of the Kerberos specification because the behavior based on these values should be understood across implementations whereas other elements need only be understood by the applications which they affect.

In the definitions that follow, the value of the ad-type for the element will be specified in the subsection number, and the value of the ad-data will be as shown in the ASN.1 structure that follows the subsection heading.

B.1. If relevant

AD-IF-RELEVANT AuthorizationData

AD elements encapsulated within the if-relevant element are intended for interpretation only by application servers that understand the particular ad-type of the embedded element. Application servers that do not understand the type of an element embedded within the if-relevant element may ignore the uninterpretable element. This element promotes interoperability across implementations which may have local extensions for authorization.

B.2. Intended for server

```
AD-INTENDED-FOR-SERVER   SEQUENCE {
         intended-server[0]     SEQUENCE OF PrincipalName
         elements[1]            AuthorizationData
}
```

AD elements encapsulated within the intended-for-server element may be ignored if the application server is not in the list of principal names of intended servers. Further, a KDC issuing a ticket for an application server can remove this element if the application server is not in the list of intended servers.

Application servers should check for their principal name in the intended-server field of this element. If their principal name is not found, this element should be ignored. If found, then the encapsulated elements should be evaluated in the same manner as if they were present in the top level authorization data field. Applications and application servers that do not implement this element should reject tickets that contain authorization data elements of this type.

B.3. Intended for application class

AD-INTENDED-FOR-APPLICATION-CLASS SEQUENCE { intended-application-class[0] SEQUENCE OF GeneralString elements[1] AuthorizationData } AD elements encapsulated within the intended-for-application-class element may be ignored if the application server is not in one of the named classes of application servers. Examples of application server classes include "FILESYSTEM", and other kinds of servers.

This element and the elements it encapulates may be safely ignored by applications, application servers, and KDCs that do not implement this element.

B.4. KDC Issued

```
AD-KDCIssued   SEQUENCE {
```

```
                ad-checksum[0]    Checksum,
                 i-realm[1]       Realm OPTIONAL,
                 i-sname[2]       PrincipalName OPTIONAL,
                elements[3]       AuthorizationData.
}
```

ad-checksum
: A checksum over the elements field using a cryptographic checksum method that is identical to the checksum used to protect the ticket itself (i.e. using the same hash function and the same encryption algorithm used to encrypt the ticket) and using a key derived from the same key used to protect the ticket.

i-realm, i-sname
: The name of the issuing principal if different from the KDC itself. This field would be used when the KDC can verify the authenticity of elements signed by the issuing principal and it allows this KDC to notify the application server of the validity of those elements.

elements
: A sequence of authorization data elements issued by the KDC.

The KDC-issued ad-data field is intended to provide a means for Kerberos principal credentials to embed within themselves privilege attributes and other mechanisms for positive authorization, amplifying the priveleges of the principal beyond what can be done using a credentials without such an a-data element.

This can not be provided without this element because the definition of the authorization-data field allows elements to be added at will by the bearer of a TGT at the time that they request service tickets and elements may also be added to a delegated ticket by inclusion in the authenticator.

For KDC-issued elements this is prevented because the elements are signed by the KDC by including a checksum encrypted using the server's key (the same key used to encrypt the ticket - or a key derived from that key). Elements encapsulated with in the KDC-issued element will be ignored by the application server if this "signature" is not present. Further, elements encapsulated within this element from a ticket granting ticket may be interpreted by the KDC, and used as a basis according to policy for including new signed elements within derivative tickets, but they will not be copied to a derivative ticket directly. If they are copied directly to a derivative ticket by a KDC that is not aware of this element, the signature will not be correct for the application ticket elements, and the field will be ignored by the application server.

This element and the elements it encapulates may be safely ignored by applications, application servers, and KDCs that do not implement this element.

B.5. And-Or

```
AD-AND-OR           SEQUENCE {
                        condition-count[0]    INTEGER,
                        elements[1]           AuthorizationData
}
```

When restrictive AD elements encapsulated within the and-or element are encountered, only the number specified in condition-count of the encapsulated conditions must be met in order to satisfy this element. This element may be used to implement an "or" operation by setting the condition-count field to 1, and it may specify an "and" operation by setting the condition count to the number of embedded elements. Application servers that do not implement this element must reject tickets that contain authorization data elements of this type.

B.6. Mandatory ticket extensions

```
AD-Mandatory-Ticket-Extensions          SEQUENCE {
                        te-type[0]      INTEGER,
                        te-checksum[0]   Checksum
}
```

An authorization data element of type mandatory-ticket-extensions specifies the type and a collision-proof checksum using the same hash algorithm used to protect the integrity of the ticket itself. This checksum will be calculated over an individual extension field of the type indicated. If there are more than one extension, multiple Mandatory-Ticket-Extensions authorization data elements may be present, each with a checksum for a different extension field. This restriction indicates that the ticket should not be accepted if a ticket extension is not present in the ticket for which the type and checksum do not match that checksum specified in the authorization data element. Note that although the type is redundant for the purposes of the comparison, it makes the comparison easier when multiple extensions are present. Application servers that do not implement this element must reject tickets that contain authorization data elements of this type.

B.7. Authorization Data in ticket extensions

```
AD-IN-Ticket-Extensions   Checksum
```

An authorization data element of type in-ticket-extensions specifies a collision-proof checksum using the same hash algorithm used to protect the integrity of the ticket itself. This checksum is calculated over a separate external AuthorizationData field carried in the ticket extensions. Application servers that do not implement this element must reject tickets that contain authorization data elements of this type. Application servers that do implement this element will search the ticket extensions for authorization data fields, calculate the specified checksum over each authorization data field and look for one matching the checksum in this

in-ticket-extensions element. If not found, then the ticket must be rejected. If found, the corresponding authorization data elements will be interpreted in the same manner as if they were contained in the top level authorization data field.

Note that if multiple external authorization data fields are present in a ticket, each will have a corresponding element of type in-ticket-extensions in the top level authorization data field, and the external entries will be linked to the corresponding element by their checksums.

C. Definition of common ticket extensions

This appendix contains the definitions of common ticket extensions. Support for these extensions is optional. However, certain extensions have associated authorization data elements that may require rejection of a ticket containing an extension by application servers that do not implement the particular extension. Other extensions have been defined beyond those described in this specification. Such extensions are described elswhere and for some of those extensions the reserved number may be found in the list of constants.

It is known that older versions of Kerberos did not support this field, and that some clients will strip this field from a ticket when they parse and then reassemble a ticket as it is passed to the application servers. The presence of the extension will not break such clients, but any functionaly dependent on the extensions will not work when such tickets are handled by old clients. In such situations, some implementation may use alternate methods to transmit the information in the extensions field.

C.1. Null ticket extension

TE-NullExtension OctetString -- The empty Octet String

The te-data field in the null ticket extension is an octet string of lenght zero. This extension may be included in a ticket granting ticket so that the KDC can determine on presentation of the ticket granting ticket whether the client software will strip the extensions field. =20

C.2. External Authorization Data

TE-ExternalAuthorizationData AuthorizationData

The te-data field in the external authorization data ticket extension is field of type AuthorizationData containing one or more authorization data elements. If present, a corresponding authorization data element will be present in the primary authorization data for the ticket and that element will contain a checksum of the external authorization data ticket extension.

D. Significant changes since RFC 1510

Section 1: The preamble and introduction does not define the protocol, mention is made in the introduction regarding the ability to rely on the KDC to check the transited field, and on the inclusion of a flag in a ticket indicating that this check has occurred. This is a new capability not present in RFC1510. Pre-existing implementation may ignore or not set this flag without negative security implications.

The definition of the secret key says that in the case of a user the key may be derived from a password. In 1510, it said that the key was derived from the password. This change was made to accommodate situations where the user key might be stored on a smart-card, or otherwise obtained independent of a password.

The introduction also mentions the use of public key for initial authentication in Kerberos by reference. RFC1510 did not include such a reference.

Section 1.2 was added to explain that while Kerberos provides authentication of a named principal, it is still the responsibility of the application to ensure that the authenticated name is the entity with which the application wishes to communicate.

Section 2: No changes were made to existing options and flags specified in RFC1510, though text was revised to make the description and intent of existing options clearer, especially with respect to the ENC-TKT-IN-SKEY option (now section 2.9.3) which is used for user-to-user authentication. New options and ticket flags added since RFC1510 include transited policy checking (section 2.7), anonymous tickets (section 2.8) and name canonicalization (section since dropped, but flags remain).

Section 3: Added mention of the optional checksum field in the KRB-ERROR message. Added mention of name canonicalization (since deleted) and anonymous tickets in exposition on KDC options. Mention of the name canonicalization case was included in the description of the KDC reply (3.1.3). A warning regarding generation of session keys for application use was added, urging the inclusion of key entropy from the KDC generated session key in the ticket. An example regarding use of the subsession key was added to section 3.2.6. Descriptions of the pa-etype-info, and pa-pw-salt preauthentication data items were added.

Changes to section 4: Added language about who has access to the keys in the Kerberos database. Also made it clear that KDC's may obtain the information from some database field through other means - for example, one form of pkinit may extract some of these fields from a certificate.

Section 5: The message specification section has undergone a major rewrite to eliminate confusion regarding different versions of ASN.1, and to highlight those areas where the Kerberos protocol does not strictly follow

ASN.1. These are changes to make the description more clear, rather than changes to the protocol.

Major changes were made to message numbers, providing a clearer interoperability path for messages that have new optional fields. The specific additions to these messages are listed specifically elswhere in this document.

A statement regarding the carrying of unrecognized additional fields in ASN.1 encoding through in tickets was added (still waiting on some better text regarding this).

Ticket flags and KDC options were added to support the new functions described elsewhere in this document. The encoding of the options flags are now described to be no less than 32 bits, and the smallest number of bits beyond 32 needed to encode any set bits. It also describes the encoding of the bitstring as using "unnamed" bits.

An optional ticket extensions field was added to support the carrying of auxiliary data that allows the passing of auxiliary that is to accompany a ticket to the verifier.

(Still pending, Tom Yu's request to change the application codes on KDC message to indicate which minor rev of the protocol - I think this might break things, but am not sure).

Definition of the PA-USE-SPECIFIED-KVNO preauthentication data field was added.

The optional e-cksum field was added to the KRB-ERROR message and the e-data filed was generalized for use in other than the KDC_ERR_PREAUTH_REQUIRED error. The TypedData structure was defined. Type tags for TypedData are defined in the same sequence as the PA-DATA type space to avoid confusion with the use of the PA-DATA namespace previously used for the e-data field for the KDC_ERR_PREAUTH_REQUIRED error.

Section 6: Section 6 has undergone a major rewrite to more redily convey how to add new encryption and checksum methods to Kerberos. New encryption methods were added. Existing methods that are in use have not changed, but their descriptions have to yield bettwer symmetry with the discussions of the new methods.

Section 7: Words were added describing the convention that domain based realm names for newly created realms should be specified as upper case. This recommendation does not make lower case realm names illegal. Words were added highlighting that the slash separated components in the X500 style of realm names is consistent with existing RFC1510 based implementations, but that it conflicts with the general recommendation of X.500 name representation specified in RFC2253.

There were suggestions on the list regarding extensions to or new name types. These require discussion at the IETF meeting. My own feeling at this point is that in the absence of a strong consensus for adding new types at this time, I would rather not add new name types in the current draft, but leave things open for additions later.

Section 8: Since RFC1510, the definition of the TCP transport for Kerberos messages was added.

Section 9: Requirements for supporting DES3-CBC-SHA1-KD encryption and HMAC-SHA1-DES3-KD checksums were added.

I would like to make support for Rijndael mandatory and for us to have a SINGLE standard for use of Rijndale in these revisions.

Y. Deprecated checksum types

Work is still needed on this section.

* More detailed list of differences from RFC 1510, to update the "Significant changes" appendix.
* Compare specs against actual implementations (e.g., choice of ivec) and make sure they match.

See end notes for other issues.

-- Ken 2001-11-20

Give me an appendix letter?

This section describes some checksum mechanisms defined in RFC 1510 but considered deprecated in this specification. While the authors believe the mechanisms currently in common use have all been included in section 6, these deprecated mechanisms may still be available in older implementations, so we include them here, modified to fit the framework outlined in 6. They are not required for an implementation to be conformant to this specification.

Y.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.

```
 associated cryptosystem des-cbc-md5, des-cbc-md4, des-cbc-crc
 get_mic                 crc32(msg)

 verify_mic              compute checksum and compare
```
The crc32 checksum algorithm is assigned a checksum type number of one (1).

Y.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.

```
 associated cryptosystem des-cbc-md5, des-cbc-md4, des-cbc-crc
 get_mic                 md4(msg)

 verify_mic              compute checksum and compare
```
The rsa-md4 checksum algorithm is assigned a checksum type number of two (2).

Y.3. 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 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".

```
 associated    des-cbc-md5, des-cbc-md4, des-cbc-crc
 cryptosystem
 get_mic       des-cbc(key XOR F0F0F0F0F0F0F0F0,
                       confounder | des-mac(key, confounder | msg, ivec=0),
                       ivec=0)

 verify_mic    decrypt, compute DES MAC using confounder, compare
```
The des-mac checksum algorithm is assigned a checksum type number of four (4).

Y.4. 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.

```
 associated cryptosystem des-cbc-md5, des-cbc-md4, des-cbc-crc
 get_mic                 des-cbc(key, md4(msg), ivec=key)

 verify_mic              compute CRC-32 and compare
```

The rsa-md4-des-k checksum algorithm is assigned a checksum type number of six (6).

Y.5. DES cipher-block chained checksum alternative (des-mac-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.

```
 associated cryptosystem des-cbc-md5, des-cbc-md4, des-cbc-crc
 get_mic                 des-mac(key, msg, ivec=key or given)

 verify_mic              compute MAC and compare
```

The des-mac-k checksum algorithm is assigned a checksum type number of five (5).

Appendix Z. Test Vectors

This section provides test vectors for various functions defined or described in section 6. For convenience, most inputs are ASCII strings, though some UTF-8 samples should be provided for string-to-key functions. Keys and other binary data are specified as hexadecimal strings.

Z.1. n-fold

The n-fold function is defined in section 6.4. As noted there, the sample vector in the original paper defining the algorithm appears to be incorrect. Here are values provided by Marc Horowitz:

```
   64-fold("012345") =
   64-fold(303132333435) = be072631276b1955

   56-fold("password") =
   56-fold(70617373776f7264) = 78a07b6caf85fa

   64-fold("Rough Consensus, and Running Code") =
```

```
   64-fold(526f75676820436f6e73656e7375732c20616e642052756e
           6e696e6720436f6465) = bb6ed30870b7f0e0

   168-fold("password") =
   168-fold(70617373776f7264) =
            59e4a8ca7c0385c3c37b3f6d2000247cb6e6bd5b3e

   192-fold("MASSACHVSETTS INSTITVTE OF TECHNOLOGY"
   192-fold(4d4153534143485653455454532049 4e5354495456544520
            4f4620544543484e4f4c4f4759) =
            db3b0d8f0b061e603282b308a50841229ad798fab9540c1b
```

Z.2. mit_des_string_to_key

The function mit_des_string_to_key is defined in section 6.5.2. We present here several test values, with some of the intermediate results. The fourth test demonstrates the use of UTF-8 with three characters. The last two tests are specifically constructed so as to trigger the weak-key fixups for the intermediate key produced by fan-folding; we have no test cases that cause such fixups for the final key.

```
UTF-8 encodings:
 eszett  C3 9F
 s-caron C5 A1
 c-acute C4 87

Test vector:

salt:     "ATHENA.MIT.EDUraeburn"  415448454e412e4d49542e4544557261656275726e
password: "password"               70617373776f7264
fan-fold result:                   c01e38688ac86c2e
intermediate key:                  c11f38688ac86d2f
DES key:                           cbc22fae235298e3

salt:     "WHITEHOUSE.GOVdanny"    5748495445484f5553452e474f5664616e6e79
password: "potatoe"                706f7461746f65
fan-fold result:                   a028944ee63c0416
intermediate key:                  a129944fe63d0416
DES key:                           df3d32a74fd92a01
```



```
salt:     "EXAMPLE.COMbuckaroo"    4558414d504c452e434f4d6275636b61726f6f
password: "penny"                  70656e6e79
fan-fold result:                   96d2d87e925c64ee
intermediate key:                  97d3d97f925d64ef
DES key:                           9443a2e532fdc4f1

salt:     "ATHENA.MIT.EDUJuri" + s-caron + "i" + c-acute
                           415448454e412e4d49542e4544554a757269c5a169c487
password: eszett                   c39f
```

```
fan-fold result:                    b8f6c40e305afc9e
intermediate key:                   b9f7c40e315bfd9e
DES key:                            62c81a5232b5e69d

salt:     "AAAAAAAA"                4141414141414141
password: "11119999"                3131313139393939
fan-fold result:                    e0e0e0e0f0f0f0f0
intermediate key:                   e0e0e0e0f1f1f101
DES key:                            984054d0f1a73e31

salt:     "FFFFAAAA"                4646464641414141
password: "NNNN6666"                4e4e4e4e36363636
fan-fold result:                    1e1e1e1e0e0e0e0e
intermediate key:                   1f1f1f1f0e0e0efe
DES key:                            c4bf6b25adf7a4f8
```

Z.3. DES3 DR and DK

These tests show the derived-random and derived-key values for the des3-hmac-sha1-kd encryption scheme, using the DR and DK functions defined in section 6.5.5. The input keys were randomly generated; the usage values are ones actually used by Kerberos.

```
key:    dce06b1f64c857a11c3db57c51899b2cc1791008ce973b92
usage:  0000000155
DR:     935079d14490a75c3093c4a6e8c3b049c71e6ee705
DK:     925179d04591a79b5d3192c4a7e9c289b049c71f6ee604cd

key:    5e13d31c70ef765746578531cb51c15bf11ca82c97cee9f2
usage:  00000001aa
DR:     9f58e5a047d894101c469845d67ae3c5249ed812f2
DK:     9e58e5a146d9942a101c469845d67a20e3c4259ed913f207

key:    98e6fd8a04a4b6859b75a176540b9752bad3ecd610a252bc
usage:  0000000155
DR:     12fff90c773f956d13fc2ca0d0840349dbd39908eb
DK:     13fef80d763e94ec6d13fd2ca1d085070249dad39808eabf

key:    622aec25a2fe2cad7094680b7c64940280084c1a7cec92b5
usage:  00000001aa
DR:     f8debf05b097e7dc0603686aca35d91fd9a5516a70
DK:     f8dfbf04b097e6d9dc0702686bcb3489d91fd9a4516b703e

key:    d3f8298ccb166438dcb9b93ee5a7629286a491f838f802fb
usage:  6b65726265726f73
DR:     2270db565d2a3d64cfbfdc5305d4f778a6de42d9da
DK:     2370da575d2a3da864cebfdc5204d56df779a7df43d9da43

key:    b55e983467e551b3e5d0e5b6c80d45769423a873dc62b30e
usage:  636f6d62696e65
DR:     0127398bacc81a2a62bc45f8d4c151bbcdd5cb788a
DK:     0126388aadc81a1f2a62bc45f8d5c19151bacdd5cb798a3e
```

```
key:    c1081649ada74362e6a1459d01dfd30d67c2234c940704da
usage:  0000000155
DR:     348056ec98fcc517171d2b4d7a9493af482d999175
DK:     348057ec98fdc48016161c2a4c7a943e92ae492c989175f7

key:    5d154af238f46713155719d55e2f1f790dd661f279a7917c
usage:  00000001aa
DR:     a8818bc367dadacbe9a6c84627fb60c294b01215e5
DK:     a8808ac267dada3dcbe9a7c84626fbc761c294b01315e5c1

key:    798562e049852f57dc8c343ba17f2ca1d97394efc8adc443
usage:  0000000155
DR:     c813f88b3be2b2f75424ce9175fbc8483b88c8713a
DK:     c813f88a3be3b334f75425ce9175fbe3c8493b89c8703b49

key:    26dce334b545292f2feab9a8701a89a4b99eb9942cecd016
usage:  00000001aa
DR:     f58efc6f83f93e55e695fd252cf8fe59f7d5ba37ec
DK:     f48ffd6e83f83e7354e694fd252cf83bfe58f7d5ba37ec5d
```

Z.4. DES3string_to_key

These are the keys generated for some of the above input strings for triple-DES with key derivation as defined in section 6.5.5.

```
salt:   "ATHENA.MIT.EDUraeburn"
passwd: "password"
key:    850bb51358548cd05e86768c313e3bfef7511937dcf72c3e

salt:   "WHITEHOUSE.GOVdanny"
passwd: "potatoe"
key:    dfcd233dd0a43204ea6dc437fb15e061b02979c1f74f377a

salt:   "EXAMPLE.COMbuckaroo"
passwd: "penny"
key:    6d2fcdf2d6fbbc3ddcadb5da5710a23489b0d3b69d5d9d4a

salt:   "ATHENA.MIT.EDUJuri" + s-caron + "i" + c-acute
passwd: eszett
key:    16d5a40e1ce3bacb61b9dce00470324c831973a7b952feb0
```

Z.5. DES3 combine-keys

PLACEHOLDER FOR NEW DATA

Footnotes:

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

1.1
Note, however, that many applications use Kerberos' functions only upon the initiation of a stream-based network connection. Unless an application subsequently provides integrity protection for the data stream, the identity verification applies only to the initiation of the connection, and does not guarantee that subsequent messages on the connection originate from the same principal.

1.2
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.

1.3
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.

2.1
Though it is permissible to request or issue tick- ets with no network addresses specified.

2.2
It is important that the KDC be sent the name as typed by the user, and not only the canonical form of the name. If the domain name system was used to find the canonical name on the client side, the mapping is vulnerable.

3.1
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.

3.2
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 a local server through successful completion of the

Client/Server exchange.

3.3

"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 is more desirable to use a truly random number generator, such as one based on measurements of random physical phenomena.

3.4

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.

3.5

Note that this can make applications based on unreliable transports difficult to code correctly. If the transport might deliver duplicated messages, either a new authenticator must be generated for each retry, or the application server must match requests and replies and replay the first reply in response to a detected duplicate.

3.6

This allows easy implementation of user-to-user authentication [8], which uses ticket-granting ticket session keys in lieu of secret server keys in situations where such secret keys could be easily compromised.

3.7

Note also 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.

3.8

If this is not done, an attacker could subvert the authentication by recording the ticket and authenticator sent over the network to a server and replaying them following an event that caused the server to lose track of recently seen authenticators.

3.9

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.

3.10

Note that for encrypting the KRB_AP_REP message, the sub-session key is

not used, even if present in the Authenticator.

3.11

Implementations of the protocol may wish to provide routines to choose subkeys based on session keys and random numbers and to generate a negotiated key to be returned in the KRB_AP_REP message.

3.12

This can be accomplished in several ways. It might be known beforehand (since the realm is part of the principal identifier), it might be stored in a nameserver, or it might be 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.

3.13

If the client selects a sub-session key, care must be taken to ensure the randomness of the selected sub-session key. One approach would be to generate a random number and XOR it with the session key from the ticket-granting ticket.

4.1

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.

4.2

See the discussion of the padata field in section 5.4.2 for details on why this can be useful.

6.1

While Message Authentication Code (MAC) or Message Integrity Check (MIC) would be more appropriate terms for many of the uses in this section, we continue to use the term "checksum" for historical reasons.

6.2

For example, a pseudo-random number generator may be seeded with a session key, but to protect the original key from any accidental weakness in the PRNG, use possibly-known data encrypted or checksummed using the key rather than using the key directly. Usage numbers in this reserved range should help avoid accidentally seeding the PRNG with a value also computed and perhaps exposed to an attacker elsewhere.

6.3

Of course, this does not apply to protocols that do their own encryption independent of this framework, directly using the key resulting from the Kerberos authentication exchange.

6.4

It should be noted that the sample vector in Appendix B.2 of the original paper appears to be incorrect. Two independent implementations from the specification (one in C by Marc Horowitz, and another in Scheme by Bill Sommerfeld) agree on a value different from that in [Blumenthal96].

6.5

Some problematic assumptions we've seen, and sometimes made, include: that a random bitstring is always valid as a key (not true for DES keys with parity); that the basic block encryption chaining mode provides no integrity checking, or can easily be separated from such checking (not true for many modes in development that do both simultaneously); that a checksum for a message always results in the same value (not true if a confounder is incorporated); that an initial vector is used (may not be true if a block cipher in CBC mode is not in use); that the key is a clever thing to use as the initial vector for CBC mode encryption (not true @@REF Bellovin paper).

6.6

Perhaps one of the more common reasons for directly performing encryption is direct control over the negotiation and to select a "sufficiently strong" encryption algorithm (whatever that means in the context of a given application). While Kerberos directly provides no facility for negotiating encryption types between the application client and server, there are other means for accomplishing similar goals. For example, requesting only "strong" session key types from the KDC, and assuming that the type actually returned by the KDC will be understood and supported by the application server.