Date: 6 November 1989
From: John Kohl, Cliff Neuman, Jennifer Steiner
To: RFC readers
Re: Kerberos Version 5 RFC, draft #2

This is the second draft of the proposed Kerberos Version 5 protocol specification RFC-style document.

We would like the readers to note several things:

We are interested in comments on whether it is appropriate to make further changes to the Kerberos protocol so that it conforms with ISO's ASN.1. The X.500 committee seems to be interested in allowing the use of Kerberos as an "external" mechanism for authentication in their directory service. For them to do this, they would want the Kerberos protocol to be ISO conformant. The advantage to us is that if Kerberos receives specific mention as an example of an "external" authentication service, it would certainly increase its appeal to organizations that take standards seriously.

Some parts of the V5 protocol draft are already taken from ASN.1 (the byte ordering, and the format for some of the field lengths). Other changes that would be required would increase the size of the messages, and because of encryption would probably affect efficiency. If comments indicate it would be worthwhile, our approach will probably be to work out an alternative V5 proposal with an encoding that conforms to ASN.1.

The protocol version number fields in the messages used by Kerberos are preceded by ASN.1 type and length information; this is intended so that future ASN.1 implementations can parse the first field and recognize a non-conformant encoding of the message. An alternative approach would make the initial field an integer (the ASN.1 protocol version number) in the ASN.1 version, and an octetstring of bytes in the non ASN.1 version. If we use this approach, we need to make sure that such an approach would allow us to interoperate (in a limited sense) with future ASN.1 encodings and implementations. Another alternative approach would put the octetstring tag and the asn1 length first, and let the length include the integer tag and the (1 byte) integer representing the encoding type. This way, the whole message (or authenticator or ticket) could be treated as a single unit. With the existing encoding, it has to be treated as two units.

We would like comments on the encoding described in this document and the alternatives proposed here; we also welcome comments or suggestions for a different encoding.

This draft specifies some implementation restrictions on the required sizes allowed for certain string elements in the protocol messages (See section 5.1 for details on how these limits are to be used). If you feel any of these limits are inappropriate (too large or too small), please send comments!.

We are still looking for a good, fast, secure cryptographic checksum for use in the KRB_SAFE exchange.

We are unsure if 2 bytes of random data are sufficient for a confounder. We may use a longer random field if necessary.

We are considering modifying the KRB_TGS_REP request to not encrypt the second ticket and authorization_data. However, we are concerned about the possible attacks on these and the response if they are only integrity-protected. If we choose not to protect these field in the request, then we would add fields to the response to allow the client to verify that the request was not modified. This is acceptable only if the response, as sent over the network, would not be useful to an attacker that had modified the request. We seek comments regarding the possible attacks and/or consequences of this approach, particularly with respect to interactions with some of the new options which are available. We seek comments regarding the possible attacks and/or the consequences of only integrity-protecting these portions of the TGS_REP.

The pseudo-code provided in appendix A is a "first pass" and not fully "debugged". We welcome comments on errors and suggestions for more or less detail there.

Please send any comments about this draft to the mailing list `krb-protocol@athena.mit.edu`.

We thank you for your interest in Kerberos, and look forward to hearing your comments.

DRAFT 2

Major changes since draft 1

This list doesn't include rewordings, typos & such.

- Principal names are arrays of strings, rather than a name,instance pair.
- Length restrictions placed on some fields.
- Integrity checksums are now considered part of the encryption function
- No longer use timestamp+1 in KRB_AP_REP.
- Drop support or recommendation of modified Juenemann Checksum as a crypto checksum.
- Direction bit in KRB_SAFE and KRB_PRIV is now placed in the 2-byte millisecond field.
- Addition of pseudo-code in appendix.

Network Working Group
Request for Comments: DRAFT 2

John Kohl
B. Clifford Neuman
Jennifer Steiner
MIT Project Athena
6 November 1989

The Kerberos Network Authentication Service

DRAFT

STATUS OF THIS MEMO

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

OVERVIEW

This DRAFT RFC describes the concepts and model upon which the Kerberos network authentication system is based. It also specifies the present proposal for version 5.

The motivations, goals, assumptions, and rationale behind design decisions are treated cursorily; they are fully described for the previous version in the Kerberos portion of the Athena Technical Plan.[1] The protocols are under review, and are not proposed as an Internet standard at this time. Comments are encouraged. Requests for additions to an electronic mailing list on Kerberos discussions, `kerberos@athena.mit.edu`, may be addressed to `kerberos-request@athena.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@athena.mit.edu`.

ACKNOWLEDGEMENTS

The Kerberos model is based on Needham and Schroeder's trusted third-party authentication scheme[3] and on modifications suggested by Denning and Sacco.[4] The original design and implementation of Kerberos versions 1 through 4 are due to two former Project Athena members, Steve Miller of Digital Equipment Corporation and Clifford Neuman of the University of Washington, 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.

1. Introduction

Kerberos provides a means of verifying the identities of principals, e.g, a workstation user or a network server, on an open (i.e. unprotected) network. This is accomplished without relying on authentication by the host operating system, without basing trust on host addresses†, without requiring physical security of all the hosts on the network, and under the assumption that packets traveling along the network can be read, modified, and inserted at will. Kerberos performs authentication under these conditions as a trusted third-party authentication service using conventional (shared secret key‡) cryptography.

---

† Note, however, that many applications use Kerberos' functions only upon the initiation of a stream-based network connection, and assume the absence of any ''hijackers'' who might subvert such a connection. Such use implictly trusts the host addresses involved.

‡*Secret* and *private* are often used interchangeably in the literature. In our usage, it takes two (or more) to share a secret, thus a shared DES key is a *secret* key. Something is only private when no one but its owner knows it. Thus, in public key cryptosystems, one has a public and a *private* key.

The authentication process proceeds as follows: A client sends a request to the authentication server (AS) requesting "credentials" for a given server. The AS responds with these credentials, encrypted in the client's key. The credentials consist of 1) a "ticket" for the server and 2) a temporary (session) encryption key. The client forwards 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 optionally authenticate the server. It may also be used to encrypt further communication between the two parties.

The implementation consists of one or more authentication servers running on physically secure hosts. The authentication servers maintain a database of principals (i.e., users and servers) and their secret (private) keys. 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.

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

Once a client has obtained credentials for a server (using either of the two methods above), it is up to the specific application to decide how they are to be used. We have implemented several methods for using the credentials. In the first, the client forwards the ticket to the server, along with information which helps to detect replays. Since the ticket is sent in the clear, and may be reused for a limited period of time, there must be some way for the server to know not only to whom the ticket was issued, but also that the principal using the ticket is the same as the principal to whom it was issued. This can be done using the session key, since no one except the requesting principal and the server know it--it is never sent over the network in the clear.

The second method for using credentials affords detection not only of replay, but also of message stream modification (MSM). This is done by including a cryptographic checksum of the client's message. The checksum is computed using the session key.

A third method provides not only authentication, but also data encryption, again using the session key.

The authentication exchanges mentioned above require read-only access to the Kerberos database. Sometimes, however, the data in the database must be modified, such as when adding new principals or changing a password. This is done using a protocol between a client and a third Kerberos server, the Kerberos Administration Server (KADM). This administration protocol is not described in this document.

Inter-Realm Operation

The Kerberos protocols are 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 exchanging an "inter-realm" key, the administrators of two realms can allow a client authenticated in the local realm to use its authentication remotely. The exchange of an inter-realm key 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 to decrypt the ticket-granting ticket, and is thus certain that it was issued by the client's local Kerberos. Tickets issued by the remote ticket-granting service will indicate that the client was authenticated in its local 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.

Although realms are typically hierarchical, intermediate realms may be bypassed to achieve inter-realm authentication through alternate authentication paths. It is important for the end service to know which realms were transited when deciding how much faith to put in the authentication process. To facilitate this decision, a field in the ticket contains the names of the realms that were involved in authenticating the client. The encoding and use of this field is described later in this document.

Proxy and Authentication Forwarding

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.

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

In order to complicate the use of stolen credentials, Kerberos tickets are typically valid from only those network addresses specifically included in the ticket. For this reason, a client wishing to grant a proxy must request a new ticket valid for the network address of the service to be granted the proxy.

Kerberos supports proxy and authentication forwarding through the combined effects of several fields in the tickets it issues. The proxiable and forwardable flags in the ticket-granting ticket indicate whether a proxy can be granted without requiring the user to enter a password again. The host address field optionally restricts the proxy to being used from a particular network address. Finally, the authorization data field allows the client to include information in the proxy restricting its use. The content and use of this field are described in greater detail in sections 2.3, 6, and 7.1.

1.1. 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.

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.

Instance The name often given to the second component of a principal identifier, or a particular principal from a group of related principals. In the latter usage, the instances are often created to partition permission for users (e.g. a user might have a "normal" instance, and a "root" instance which has different privileges†) or to impose a naming convention on service key names (e.g. for a particular service, the instance(s) identifies the host machine(s) on which that service is provided and the principal identifier of the server).

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

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).

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

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

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

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

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.

---

†Note that these privileges are determined by access controls applied by application servers; the instance field does not carry any inherent privileges.

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

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

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

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

2. Message Exchanges

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

2.1. The Authentication Service (AS) Exchange

This section describes one interaction between a client and the Kerberos Authentication Server. This exchange is usually initiated by a client when it wishes to obtain authentication credentials for a given server. 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 obtain credentials for other servers (see section 2.3) without requiring further use of the client's secret key. This exchange is used to request credentials for services which must not be mediated through the Ticket-Granting Service, but rather require a principal's secret key, such as the password-changing service†.

The 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 section 7.2.

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. 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_AS_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 (the lack of encryption in the KRB_ERROR message thwarts the ability to detect replays).

It should be noted that 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.

2.1.1. Generation of KRB_AS_REQ message

The client may specify a number of options in the initial request. Among these options are whether the requested ticket is to be renewable, proxiable, or forwardable; whether it should be postdated or allow postdating of derivative tickets; and whether a renewable ticket will be accepted in lieu of a non-renewable ticket if the requested ticket expiration date cannot be satisfied by a non-renewable ticket (due to

† 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.

configuration constraints; see section 4).

The client prepares the KRB_AS_REQ message containing a field of desired options, the desired start time (after which the ticket should be valid), the desired expiration time (after which the ticket should be invalid), the desired encryption type, the client's name, and the server's name, and sends it to the KDC.

2.1.2. 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 7.1. The contents of the ticket are determined as follows.

2.1.3. 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 server cannot accomodate the requested encryption type, an error message with code KDC_ERR_ETYPE_NOSUPP is returned. Otherwise it generates a "random" session key†.

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

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

- The expiration time requested in the KRB_AS_REQ message
- The ticket's start time plus the maximum allowable lifetime associated with the client principal (The authentication server's database includes a maximum ticket lifetime field in each principal's record; see section 4).
- The ticket's start time plus the maximum allowable lifetime associated with the server principal.
- The ticket's start time plus the lifetime set by the policy of the local realm.

If the requested expiration time is less than a site-determined constant greater than the start time determined as above, an error message with code KDC_ERR_NEVER_VALID is returned (the 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). 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 7).

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:

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

POSTDATED, FORWARDABLE, PROXIABLE, MAY-POSTDATE, RENEWABLE, DUPLICATE-SKEY. If the new ticket is postdated (the start time is in the future), its POSTDATED flag will also be set.

If all of the above succeed, the server formats a KRB_AS_REP message (see section 7.2), encrypts the ciphertext part in the client's key using the requested encryption method, and sends it to the client.

2.1.4. Generation of KRB_ERROR message

Several errors can occur, and the Authentication Server responds by returning an error message, KRB_ERROR, to the client. The error message contents and details are described in Section 7.7.

2.1.5. 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 (to prevent blatant attacks by an attacker responding with a response to a completely different request). It decrypts the encrypted part of the response using its secret key, verifies that the "ctime" in the resp_cipher matches the timestamp it supplied in its request (to prevent replays). It also verifies that the "sname" and "srealm" in the response match those in the ticket (to help prevent an attacker from easily substituting some other ticket in the response), and that the host address field in the response matches the request (to guard against modification of the addresses in the request). It then stores the ticket, session key, start and expiration times, and other information for later use. The "key_exp" field from the resp_cipher may be checked to notify the user of impending key expiration (the client program could then suggest remedial action, such as a password change).

2.1.6. 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.

2.2. The Client/Server (CS) Authentication Exchange

This exchange is used by network applications to authenticate the client to the server and vice versa. The client must have already acquired a ticket/session key pair for the server using the AS or TGS exchange. The formats for the messages described in this section can be found in section 7.3.

2.2.1. The KRB_AP_REQ message

The KRB_AP_REQ contains authentication information which can be the first message, or the first part of a message, in an authenticated transaction. It contains a ticket and an authenticator, and some additional bookkeeping information (see section 7.3 for the exact format). The KRB_AP_REQ message is referred to elsewhere as the authentication header.

2.2.2. Generation of a KRB_AP_REQ message

When a client wishes to initiate authentication to a server, it creates a KRB_AP_REQ message by obtaining (either through a cache, the AS exchange, or the TGS exchange) a ticket and session key for the desired service. It then creates a new Authenticator (taking the system time, its name, possibly an application-protocol specific checksum, and the network layer address in use), and bundles together the ticket, authenticator, and associated information, and transmits the message to the server.

2.2.3. 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 must 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 7.7.

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 is 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. 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 indicates a failed integrity check, the KRB_AP_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. 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 the searched for an address matching the operating-system reported address of the client. If no match is found, the KRB_AP_ERR_BADADDR error is returned.

If the local (server) time and the client time in the authenticator differ by more than the allowable clock skew (e.g., 5 minutes), the KRB_AP_ERR_SKEW error is returned. If the server name along with the client name, time and millisecond fields from the Authenticator match any recently-seen such tuples, the KRB_AP_ERR_REPEAT error is returned†. The server must remember any authenticator presented within the allowable clock skew, so that a replay attempt is guaranteed to fail. If a server loses track of any authenticator presented within the allowable clock skew, it must reject all requests until the clock skew interval has passed. This assures that any lost or re-played authenticators will fall outside the allowable clock skew and can no longer be successfully replayed (If this is not done, an attacker could conceivably record the ticket & authenticator sent over the network to a server, then disable the client's host, pose as the disabled host, and replay the ticket & authenticator to subvert the authentication.).

The age of the ticket is computed: 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, 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.

2.2.4. 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 must be encapsulated in the application protocol if its "raw" form is not acceptable to the protocol. The timestamp and millisecond field used in the reply must be the client's timestamp and millisecond field (as provided in the authenticator)‡. The timestamp and millisecond field of the message are encrypted in the session key extracted from the ticket.

In both the one-way and mutual authentication exchanges, the peers should take care not to send

---

†Note that the rejection here is restricted to authenticators from the same principal to the same server. Other client principals communicating with the same server principal should not be have their authenticators rejected if the time and millisecond fields happen to match some other client's authenticator.

‡In the Kerberos version 4 protocol, the timestamp in the reply was the client's timestamp plus one. This was originally thought necessary since it was necessary in the Needham & Schroeder protocol. However, it was only necessary there because the message formats were such that a reply with an identical timestamp could easily be generated by an attacker watching the exchange without knowledge of the proper encryption keys. The Kerberos version 5 protocol messages are constructed in such a way that such extraction is not possible without knowledge of the proper encryption keys.

sensitive information to each other without proper protection (e.g. encryption).

2.2.5. Receipt of KRB_AP_REP message

If a KRB_AP_REP message is returned, the client uses the session key to decrypt the message, and verifies that the timestamp and msec fields match the Authenticator it sent to the server. If they match, then the client is assured that the server is genuine.

2.2.6. 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 key will be implicit in the protocol; in others the method of use must be chosen from a vast array of alternatives. We leave the protocol negotiations of how to use the key (e.g. selecting an encryption or checksum type) to the application programmer; the Kerberos protocol does not constrain the implementation options.

2.3. The Ticket-Granting Service (TGS) Exchange

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.). Unlike the AS exchange, 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 is used. Once the ticket-granting ticket or renewable ticket has expired the AS exchange must be repeated.

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 TGS request 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 free-form sequence of bytes to be sealed in the ticket for authorization use by the end server, or a second ticket (the use of which is 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. 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 (the lack of encryption in the KRB_ERROR message thwarts the ability to detect replays).

2.3.1. Generation of KRB_TGS_REQ message

Before sending a request to the ticket granting service, the client must determine in which realm the end server is registered†. If the client does not already possess a ticket granting ticket for the appropriate realm, then one must be obtained. This is first attempted by requesting a ticket granting ticket for the destination realm from the local Kerberos server. If this does not work, the the request must be made to the Kerberos server for a realm higher in the hierarchy. This request will itself require a ticket granting ticket for the intermediate realm which can be obtained by recursively applying these directions.

Once the ticket granting ticket for the appropriate realm has been obtained, the client determines the

---

†This can be accomplished in several ways. Presently, this information is obtained by looking in the krb.realms file, but the information is better suited for storage in a nameserver. However, there is a danger of being spoofed if the nameservice providing the realm name is not authenticated.

names of the Kerberos servers for the given realm (either through a nameserver, or using the krb.conf file).

As in the AS exchange, the client may specify a number of options in the TGS request. The client prepares the KRB_TGS_REQ message, providing an authentication header, and including the same fields as used in the KRB_AS_REQ message, along with two optional fields: the authorization_dat field for end-server use and an additional ticket required by some options. Once prepared, the message is sent to a Kerberos server for the destination realm.

2.3.2. Receipt of KRB_TGS_REQ message

The TGS request is processed in a manner similar to the AS request, but there are many additional checks to be performed. The user-supplied checksum in the Authenticator provided in the authentication header of the KRB_TGS_REQ message must be verified against the decrypted contents of the message, and the message rejected if the checksums do not match.

2.3.3. 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 set to KRB_TGS_REP. The detailed specification is included in section 7.2.

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 request specifies an endtime, then the endtime of the new ticket is 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 end server and the maximum life for the local realm. 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 FORWARDING option has been specified, 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 zero, then the start time of the ticket is set to the authentication server's current time. If it is non-zero but indicates a time in the past, it is treated as zero. If it is non-zero and indicates a time in the future, but the POSTDATED option has not been specified, 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.

If the DUPLICATE-SKEY option has been specified, and if a second ticket has been included in the request, and if the second ticket has the DUPLICATE-SKEY flag set, then the KDC will decrypt the second ticket using the key of the server for which it was issued, check to make sure that the principal to whom the second ticket was issued matches the one making the request, and if so it will use the session key from the second ticket as the session key for the new ticket. It will also set the DUPLICATE-SKEY flag on the new ticket†.

If the ENC-TKT-IN-SKEY option has been specified, and if a second ticket has been included in the request, then the KDC will decrypt the second ticket using the key for the server to which it was issued,

---

†One of the purposes of the Kerberos protocol is to securely exchange encryption keys. While it is possible for a user to securely exchange a single key with more than one other principal on top of the Kerberos protocol without using the DUPLICATE-SKEY feature, leaving the design of the mechanism to the application programmer can be error prone. By providing this functionaility within Kerberos, we make sure it is done right, and we make it known which keys have been passed on. If a key issued by Kerberos is passed on by an application (outside of the Kerberos protocol), the fact that it was passed on might not be known by other apllications, and a breach of security might result.

verify that it is a ticket-granting ticket, and use the session key from the second ticket to encrypt the new ticket it will issue instead of encrypting the new ticket in the key of the server for which it is issued‡.

If the name of the server in the ticket that is presented to the KDC as part of the authenticator is not that of the ticket-granting server itself, and the server is registered in the realm of the KDC, and the RENEW, VALIDATE, or PROXY options are specified in the request, then the KDC will decrypt the ticket in the authenticator using the key of the server to which it was issued, check that the RENEWABLE flag is set or the starttime has passed and the INVALID flag is set (respectively), check the renew_till field if appropriate, and issue a new ticket, either a renewal or a valid postdated ticket.

Whenever a request is made to the ticket-granting server, the presented ticket is checked against a hot-list of tickets which have been canceled. In this way, a stolen ticket-granting ticket or renewable ticket can not be used to gain additional tickets (renewals or otherwise) once the theft has been reported. Any normal ticket obtained before it was reported stolen will still be valid (because they require no interaction with the KDC), but only until their normal expiration time.

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

The ciphertext part of the response in the KRB_TGS_REP message is encrypted in the session key from the ticket-granting ticket instead of the client's secret key. Furthermore, the client's key's expiration date and the key version number fields are zeroed 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.

2.3.4. 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 private key.

2.4. 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 checksum of the user data and some control information. The checksum is cryptographically generated using the session key.

2.4.1. 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 will usually be some sort of cryptographic one-way hash function (such as the XXX checksum algorithm specified in section 3), seeded with an encryption key (usually the session key). Different algorithms may be selected by changing the checksum type in the message. Note that any checksum used should be careful not to reveal the session key.

After computing the checksum, the client then transmits the information and checksum to the recipient in the message format specified in section 7.5.

---

‡ This allows easy implementation of the Davis & Swick proposal[5] to use ticket-granting ticket session keys in lieu of secret server keys in situations where such secret keys could be easily compromised.

2.4.2. 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. Next the application verifies that the message length contained in the message matches the operating system's report of the message size received. A mismatch generates a KRB_AP_ERR_MODIFIED error. The application's report of the sender's address is compared against the address in the message; a mismatch generates a KRB_AP_ERR_BADADDR error. Then the time-stamp and msec fields in the message are checked to insure they are current and not replayed. If they are not current, a KRB_AP_ERR_SKEW error is generated. If they are a replay, a KRB_AP_ERR_REPEAT error is generated. The most significant bit of the millisecond field is used to encode the direction of the message (This bit is used because it can never be set as part of the encoding of a millisecond value, since such values are restricted to be less than 1000.). If the sender's network layer address is greater than the receiver's address, then the bit is set (an ordering on the addresses is specified with the specification of the encoding of the addresses, in section 5.3), otherwise it is reset. If the direction bit is set incorrectly for this message, a KRB_AP_ERR_REPEAT 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 returned.

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

2.5. 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.

2.5.1. 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 7.6) and encrypts them under an encryption key (usually the session key). It then transmits the information and some "envelope" information to the recipient.

2.5.2. 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, respectively. Next the application verifies that the message length contained in the message matches the operating system's report of the message size received. A mismatch generates a KRB_AP_ERR_MODIFIED error. The application then decrypts the encrypted data and processes them. If the length encoded in the decrypted user data is greater than the remaining length of decrypted data, a KRB_AP_ERR_MODIFIED error is generated (this ususally indicates decryption with the wrong key). The application's report of the sender's address is compared against the address in the message; a mismatch generates a KRB_AP_ERR_BADADDR error. Then the timestamp and msec fields in the message are checked to insure they are current and not replayed. If they are not current, a KRB_AP_ERR_SKEW error is generated. If they are a replay, a RD_AP_REPEAT error is generated. The most significant bit of the msec field is used to encode the direction of the message. If the sender's network layer address is greater than the receiver's address, then the bit is set (an ordering on the addresses is specified with the specification of the encoding of the addresses, in section 5.3), otherwise it is reset. If the direction bit is set incorrectly for this message, a KRB_AP_ERR_REPEAT 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. Encryption

The Kerberos protocols described in this document are designed to use stream encryption ciphers, which can be simulated using commonly available block encryption ciphers, such as the Data Encryption Standard,[6] in conjunction with block chaining and checksum methods.[7] 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 private key is used to verify the authenticity of a principal.

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

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

3.1. Cryptographic checksums

XXX need some quick crypto cksum here.

For applications that require a more trustworthy cryptographic checksum (at the cost of a serious performance degradation), the DES cipher block chain checksum should suffice.

3.2. Checksums

Some encryption systems use a block-chaining method to improve the integrity characteristics of the ciphertext. However, these chaining methods often don't provide an integrity check upon decryption. Such systems (such as DES in CBC mode) must be augmented with a checksum of the plaintext which can be verified at decryption and used to detect any tampering or damage. If any damage is detected, the decryption routine is expected to return an error indicating the failure of an integrity check.

The protocol messages only specify what fields are to be encrypted, and make no explicit requirements of a checksum. Each encryption type is expected to provide and verify an appropriate checksum. This checksum is to be encoded in the "PAD" area of the messages (note: this may necessitate an extra PAD block, depending on the encryption blocksize, the checksum size, and the plaintext length). Section 5.2.3 specifies the currently defined encryption types, their uses of checksums, and their padding requirements.

4. The Kerberos Database

The Kerberos server must have access to a database containing the names and secret keys of principals to be authenticated†.

---

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

4.1. Database contents

A database entry should contain at least the following fields:

Field	*Value*
name	Principal's name
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 first field is a string array representing the principal's name. 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 2.1 for a description of how these lifetimes are used in determining the lifetime of a given Ticket.)

If a server is to use a single database to serve several realms, the principal record should also include a realm field.

When a 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. Text that is encrypted in a principal's key is always tagged with the version of the key that was used for encryption.

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 (XXX the rest of the fields are 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. The size of the version number field in the database is an implementation issue, but only 8 bits are assigned to this field in the protocol. As such, all active keys for a given principal must have a key version number that falls into a contiguous range of 256. [One easy way to achieve this is to take the Kerberos database's key version number modulo 256, and use the result for the key version number in the protocols].

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 name

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

After an entry's 'expiration' date has passed, the KDC will return an error to any client attempting to gain tickets as or for the principal. (A database may want to maintain two expiration dates: one for the principal, and one for the principal's current key. This allows password aging to work independently of the principal's expiration date. However, due to the limited space in the responses, the KDC must combine the key expiration and principal expiration date into a single value called "key_exp", which is used as a hint to the user to take administrative action.)

The 'attributes' field is a bitfield used to govern the operations involving the principal. This field might be useful in conjunction with user registration procedures or for site-specific policy implementations (Project Athena currently uses it for their user registration process controlled by the system-wide database service, Moira.[8]). 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, disallow issuing duplicate session key tickets.

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 (more detail can be found in section 6).

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 related to them:

- The minimum supported lifetime (used to determine whether the KDC_ERR_NEVER_VALID error should be returned)
- 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)
- XXX

5. Notation

Numbers are given in decimal unless otherwise indicated.

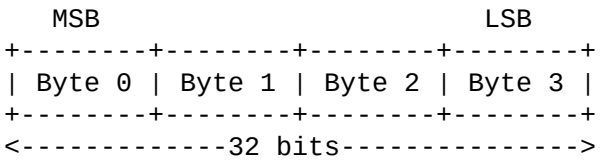
We assume 8-bit bytes. The words "byte" and "octet" are used synonymously. An octet is represented as follows:

```
 01234567
+--------+
|        |
+--------+
<-8 bits->
```

The most significant bit (msb) is bit 0; the least significant bit is bit 7.

Byte order

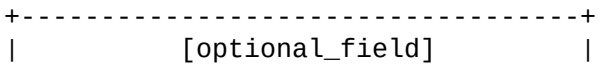
Fields which span more than one octet and represent a single numerical value are always shown in ''big-endian'' byte order (the standard Internet and ISO ASN.1 network byte order):



The most significant byte (MSB) is Byte 0; the least significant byte (LSB) in this diagram is Byte 3.

Optional fields

Some of the protocol messages have optional fields; they are labeled with square brackets surrounding the field name to indicate that they are optional:



Octet values

Some octet values are specified in a diagram by showing all eight bits in MSB order.

To avoid tedious bit-wise specification of octets, some of the following examples will specify the value of an octet in decimal (no leading digits) or hexadecimal (in the form 0xYY, where YY are the hexadecimal digits). In such cases, the value will be centered in the box around the octet. If the value being specified spans multiple octets, it will be displayed with the appropriate number of hexadecimal or decimal digits centered in those octets.

5.1. Field types

Each packet is described in terms of a table of its fields and a diagram. The table gives the length, type, label, and meaning of each field, for example:

Length	*Type*	*Label*	*Value*
1 octet	ui_1	pvno	protocol version number
1 octet	type	type	message type
4 octets	ui_4	error	error code
<= 128 octets	string	err_text	error text

The "Length" column gives the number of octets in the field. If a length is given as "<= ’y’ octets", then the length of the field is variable, and the Kerberos version 5 protocol does not specify a limit on its length. However, implementations may restrict the length, but such implementations are required to support a length of at least ’y’ octets (this length encompasses the entire encoding of the field contents, including any length indicators and type fields). Implementors should note that if their implementations generate such a field with length greater than ’y’ octets, then the protocol message containing such a field may not be accepted by some implementations. If an implementation is rejecting a message because of field length restrictions, it should use the KRB_ERR_FIELD_TOOLONG error code.

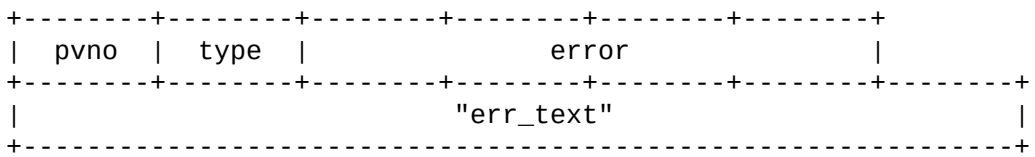
The absolute length of such a field is the length of the data plus the number of octets needed to encode its length as specified for the type bytes_asn1 (described below).

The "Type" column refers to a type described in this section.

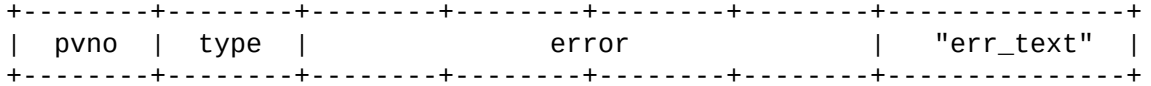
The "Label" refers to the field’s label in the diagram.

The "Value" gives the meaning of the field.

A diagram for the table above is:



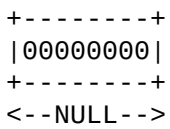
Since many fields in the Kerberos protocols are of variable length, the layout of the corresponding diagram is somewhat arbitrary. For example, the "err_text" field above is a variable-length string, so the table above could also be depicted as:



Variable-length fields which are not strings are (usually? XXX) shown in diagrams enclosed in 'single quotes'. Strings are shown in "double quotes".

5.1.1. NULL

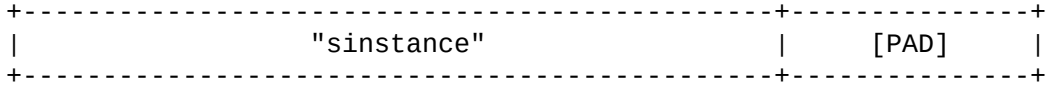
A null octet, or NULL, is an octet with 8 zero bits:



It is used to pad fields to block boundaries for encryption.

5.1.2. PAD

Some messages include variable-length fields. Block encryption ciphers require that their input and output be multiples of some block size. In these cases, a field of NULL octets is used to fill up sections of messages to be encrypted to the next multiple of the block size. This type of field is called a PAD. In the diagram representation, its label is placed in brackets to indicate that it may be of zero length.

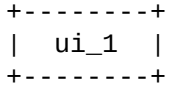


5.1.3. Unsigned Integers

Fields of unsigned integers of length 1, 2, and 4 octets are used.

ui_1

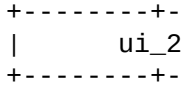
A ui_1 field consists of one octet representing an unsigned integer:



This type of field is used for some protocol version numbers, key version numbers, some length fields, and the millisecond field of a timestamp.

ui_2

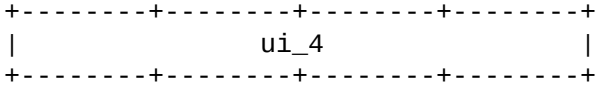
Some data lengths are given by two octets representing an unsigned integer:



The ui_2 field is used, for example, to indicate the encryption type in use in a KRB_KDC_REP message.

ui_4

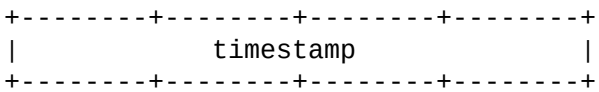
Some fields are represented by an unsigned integer of 4 octets:



This type of field is used, for example, for the 'error' field in the KRB_ERROR message to encode an error code.

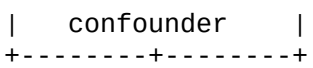
timestamp

A "timestamp" is a special case of a ui_4 field, used to indicate the date and time. The time is represented as Internet time. (Internet time is the number of seconds since 00:00:00 UTC, 1 January 1900.†)



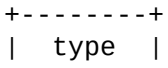
confounder

A "confounder" is a special case of a ui_2 field, used to introduce randomness into the beginning of encrypted text. This randomness makes chosen- and known-plaintext attacks more computationally intensive for most cryptosystems that will be used with Kerberos.



type

Message types are encoded in a single unsigned octet, "type". The least significant bit of all message types (but NOT other types) is zero (0) [for historical compatibility]. The message types are therefore multiples of two.



kvno

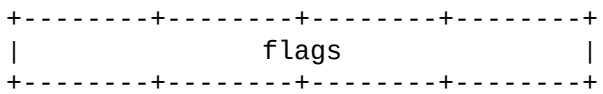
Key version numbers are maintained at the KDC in the Kerberos database. The initial version of a key is 1; subsequent versions are incremented by 1. For example, if a principal has changed its key three times, the current key will have a key version number of 4. A key version number, or "kvno" is represented as a single unsigned octet.

flags

A 32-bit (4-octet) bit field of flags (also called options) is used in a Ticket and in KDC requests/responses to indicate various options or modes of operation.

---

†The Internet timestamp encoding used here encodes a given time with an integer 2208988800 seconds greater than the timestamps used in Kerberos version 4 (which were standard UNIX timestamps, the number of seconds since 00:00:00 UTC, 1 January 1970).

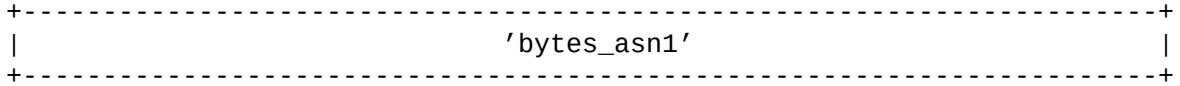


5.1.4. ASN.1 Byte vectors (bytes_asn1)

Some fields contain data which are octet strings encoded as a length sub-field followed by the contents. The length sub-field is encoded according to ASN.1 definite form (ISO 8825:1987(E), section 6.3.3) (this is an excerpt with the bit order changed to be consistent with our numbering, i.e. most significant bit is bit 0):

> If the length of the contents is 127 or less, the length sub-field is a single octet in which bit 0 is zero and bits 1 to 7 encode the number of octets in the contents sub-field (which may be zero), as an unsinged binary integer with bit 1 as the most significant bit.
>
> If the length of the contents is greater than 127, then the length sub-field consists of an initial octet and one or more subsequent octets. The initial octet shall be encoded as follows: a) bit 0 shall be one; b) bits 1 to 7 shall encode the number of subsequent octets in the length sub-field, as an unsigned binary integer with bit 1 as the most significant bit; c) the value 11111111(base 2) shall not be used. Bits 0 to 7 of the first subsequent octet, followed by bits 0 to 7 of the second subsequent octet, followed in turn by bits 0 to 7 of each further octet up to and including the last subsequent octet in the length sub-field, shall be the encoding of an unsigned binary integer equal to the number of octets in the contents sub-field.

Such fields are referred to in tables as type bytes_asn1. In diagrams, fields of this type have their names enclosed in 'single quotes' (since they are of variable length), and the octet delimiters '+' are missing:

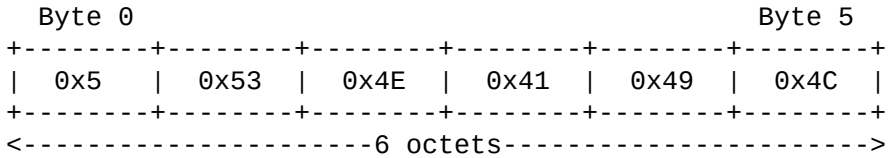


5.1.5. ASN.1 lengths

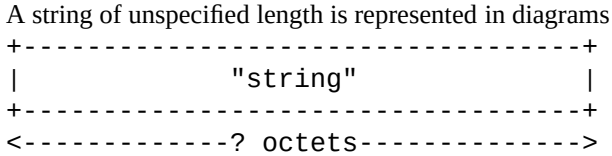
Some fields use the ASN.1 length encoding described above as a separate sub-field to denote the total length of a field.

5.1.6. Strings

Strings are fields of type bytes_asn1. Some implementations may restrict them to the short form (i.e. 127 bytes of data) of encoding. The string contents are encoded in the ISO Latin 1 character set (see ISO 8859-1)†. For example, the string "SNAIL" which has the encoding:



A string of unspecified length is represented in diagrams as:



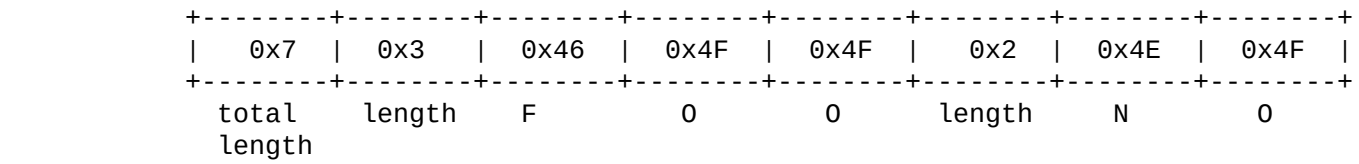
where "string" is a descriptive label. Note that the label is placed in "double quotation marks", and the octet delimiters '+' are missing. Strings are used to represent the name, instance, or realm of a Kerberos principal and error messages.

---

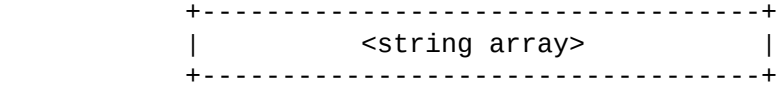
†The first 128 characters in this encoding are identical to the 7-bit ASCII encoding.

5.1.7. String Arrays

String arrays are encoded using a total length (which includes the length of all the strings plus their length encodings) followed by the string encodings in successive octets. For example, the array "FOO","NO" has the encoding:



A string array is represented in diagrams with slanted braces around the name:

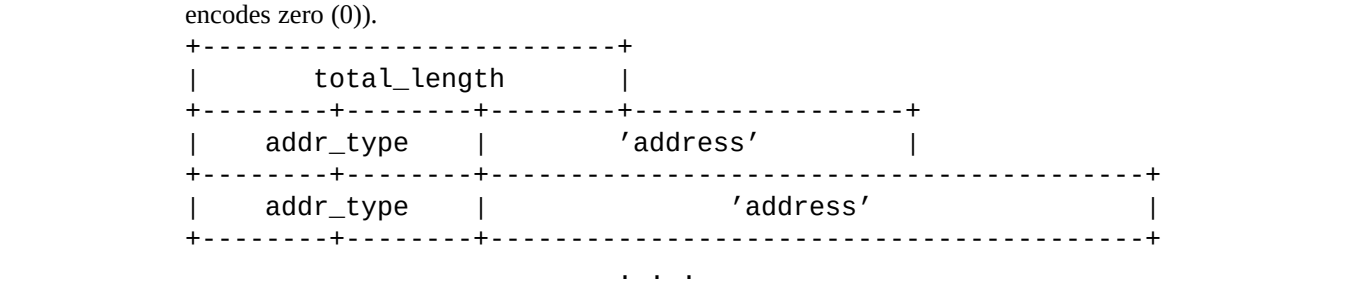


5.1.8. Host Addresses

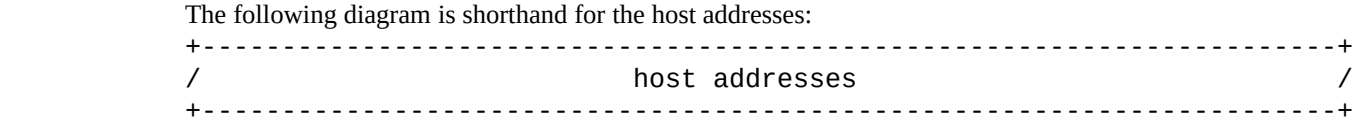
Host address fields contain zero or more network layer addresses for those hosts from which a ticket may be used. It is a compound field, consisting of the total length of the addresses' encodings and the addresses themselves. Each address is preceded by a type. This encoding is referred to as type 'hostaddrs'.

Length	*Type*	*Label*	*Value*
variable	asn1_length	total_length	Total length of network addresses
2 octets	ui_2	addr_type	Type of this address
variable	bytes_asn1	address	The address itself

The last two fields are repeated until the length is consumed (note that they may not be present if the length encodes zero (0)).



The following diagram is shorthand for the host addresses:



5.2. Predefined Data Types

This section specifies the encodings and types for encryption keys, host addresses, and other types where part of the encoding has been specified independently from the Kerberos protocol.

5.2.1. Host address types

All the values for the host address type with the most significant bit set (1) are reserved for local use. All the values with the most significant bit reset (0) 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 Distribution of Unix. They can be found in <sys/socket.h> with symbolic names AF_xxx (where xxx is an abbreviation of the address family name).

The example diagrams below show the encoding of the entire address field, which (as the addresses are encoded as type bytes_asn1) includes the length encoding as well as the address encoding.

Internet addresses

Internet addresses are 32-bit (4-octet) quantities, encoded in MSB order. The type of internet addresses is two (2). Example: the following encodes the address "18.72.0.1" [This 'dot-notation' specifies each octet of the address, from most significant to least significant, in decimal]:

```
+--------+--------+--------+
|     0x0002      |    4   |
+--------+--------+--------+--------+
|  0x12  |  0x48  |  0x00  |  0x01  |
+--------+--------+--------+--------+
```

The ordering relation between Internet addresses is determined by treating the addresses as four-octet unsigned integers with the MSB of the integer equal to the MSB of the address and comparing them as integers (e.g. 18.72.0.1 is treated as 0x12480001). If the addresses are equal, then if either UDP or TCP ports are in use, the port numbers should be treated as two-octet unsigned integers, and compared; the result of that comparison is then used as the result of the comparison of the addresses.

CHAOSnet addresses

CHAOSnet addresses are 16-bit (2-octet) quantities, encoded in MSB order. The type of CHAOSnet addresses is five (5). Example: the following encodes the address "044215" [CHAOSnet addresses are usually denoted in octal]:

```
+--------+--------+--------+--------+--------+
|     0x0005      |    2   |01001000|10001101|
+--------+--------+--------+--------+--------+
```

The ordering relation between CHAOSnet addresses is determined by treating the addresses as two-octet unsigned integers with the MSB of the integer equal to the MSB of the address, and comparing them as integers (e.g. 044215 would be less than 055161).

ISO addresses

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

```
+--------+--------+-----------------+-----------------+
|     0x0007      | length encoding |     address     |
+--------+--------+-----------------+-----------------+
```

The ordering relation between ISO addresses is determined by comparing each octet of the address, in encoding order, until a difference is encountered. The result of the comparison is the result of the comparison of the last octets or the first pair of differing octets, whichever comes first.

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). Example: the following encodes the address "08:00:2b:00:01:02" [This 'colon-notation' specifies each octet, from most significant to least significant, in hexadecimal]:

```
+--------+--------+--------+
|     0x0008      |    6   |
+--------+--------+--------+--------+--------+--------+
|  0x08  |  0x00  |  0x2b  |  0x00  |  0x01  |  0x02  |
+--------+--------+--------+--------+--------+--------+
```

The ordering relation between XNS addresses is determined by comparing each octet of the address, in encoding order, until a difference is encountered. The result of the comparison is the result of the comparison of the last octets or the first pair of differing octets, whichever comes first.

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). Example: the following encodes node 33, network 1320:

```
+--------+--------+--------+
|     0x0010      |    3   |
+--------+--------+--------+
|  0x21  |  0x05  |  0x28  |
+--------+--------+--------+
```

The ordering relation between DDP addresses is determined by comparing each octet of the address, in encoding order, until a difference is encountered. The result of the comparison is the result of the comparison of the last octets or the first pair of differing octets, whichever comes first.

5.2.2. Encryption key types

All the values for the encryption key type with the most significant bit set (1) are reserved for local use. All the values with the most significant bit reset (0) are reserved for officially assigned type fields and interpretations.

The example diagrams below show the encoding of the entire encryption key field, which (as the keys are encoded as type bytes_asn1) includes the length encoding as well as the key encoding.

NULL Key

If no encryption is in use, the encryption system is said to be the NULL encryption system. An encryption key in the NULL encryption system has type zero (0), and length zero (0). Example (remember that encryption key encodings are of type bytes_asn1, so they encode their own length):

```
+--------+--------+--------+
|     0x0000      |    0   |
+--------+--------+--------+
```

DES Key

A DES encryption key is 8 octets of data (56 bits of key, plus 8 parity bits). A DES encryption key has type one (1). Example:

```
+--------+--------+--------+
|     0x0001      |    8   |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                   DES key (64 bits/8 octets total)                    |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

Lucifer Key

A Lucifer[9] encryption key is 128 bits (16 octets) of data. A Lucifer encryption key has type two (2). Example:

```
+--------+--------+--------+
|     0x0002      |   16   |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                            Lucifer key                                |
|                     (128 bits/16 octets total)                        |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

5.2.3. Encryption system types

All the values for the encryption system type with the most significant bit set (1) are reserved for local use. All the values with the most significant bit reset (0) are reserved for officially assigned type fields and interpretations.

NULL system

If no encryption is in use, the encryption system is said to be the NULL encryption system. The NULL encryption system does not embed a checksum in the pad bytes.

The NULL encryption system has type zero (0). The blocksize of this cryptosystem is one (1) octet.

DES in CBC mode with CRC-32 checksum

When the DES is used in CBC mode with a CRC-32 checksum (described in ISO 3309[10] and many other places) of the plaintext embedded in the last four octets of the pad bytes (before encryption), the encryption type is one (1).

The CRC-32 checksum is computed over the plaintext, including the checksum or pad octets. The checksum octets are to be treated as zeroes (0) when computing the checksum.

The blocksize of this cryptosystem is eight (8) octets. The checksum requires a pad length of at least four (4) octets (i.e. acceptable pad field lengths are between 4 and 11 bytes, inclusive).

Lucifer system with CRC-32 checksum

When the Lucifer encryption system is used in XXX mode with a CRC-32 checksum embedded in the last four octets of the pad bytes (before encryption), the encryption type is two (2).

The blocksize of this cryptosystem is sixteen (16) octets. The checksum requires a pad length of at least four (4) octets (i.e. acceptable pad field lengths are between 4 and 19 bytes, inclusive).

The CRC-32 checksum is computed over the plaintext, including the checksum or pad octets. The checksum octets are to be treated as zeroes (0) when computing the checksum.

5.2.4. Checksum types

All the values for the checksum type with the most significant bit set (1) are reserved for local use. All the values with the most significant bit reset (0) are reserved for officially assigned type fields and interpretations.

The checksum types specify only the type of checksum; the length of the checksum is either explicitly stated in the use of the checksum (e.g. as part of an encryption system type) or is encoded with the checksum itself in a bytes_asn1 encoding.

CRC-32

The CRC-32 checksum has checksum type one (1).

XXX Checksum

The XXX Checksum (described in section 3) has checksum type two (2).

Xerox Secure Hash Function

The Xerox Secure Hash Function[11] has checksum type three (3).

DES cipher-block-chaining checksum (MAC)

The DES cipher-block-chaining checksum operation, known as the Message Authentication Code (MAC), has checksum type four (4).

6. Field Descriptions

Below is an alphabetical summary of the labels and descriptions of fields used in the protocol messages.

addresses This field is included in the initial request for tickets, and optionally included in requests to 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 workstation. 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. The type of this field is hostaddrs; its encoding is specified in section 5.1.8.

ap_options This field, of type flags, 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). Bit 0 is the most significant bit.

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.

asn1_header

The asn1_header field is used to allow compatibility with future implementations using alternate (ASN.1) encodings of the protocol messages. For the encoding specified in this document, its first four bytes will always be (hexadecimal) 0x02, 0x01, 0x00, 0x04:

```
+--------+--------+--------+--------+-----------------------------------+
|  0x02  |  0x01  |  0x00  |  0x04  |      ASN.1 Length encoding        |
+--------+--------+--------+--------+-----------------------------------+
  tag:     length  contents  tag:      length according to ISO 8825:1987(E)
integer   (1 byte)  (zero)  octetstring       clause 6.3.3
```

The remaining octets of the asn1_header will specify the length of the remainder of the message using the definite form of the ASN.1 length octets encoding (see below, under "ASN.1 Byte vectors").

authenticator

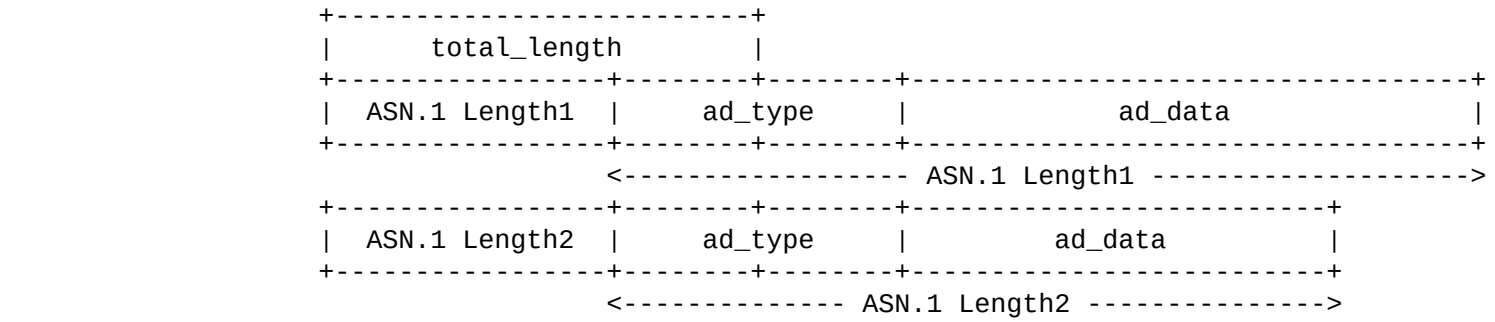
This field appears in the KRB_AP_REQ message and contains the authenticator. Its encoding is described in section 7.1.2.

authenticator_vno

This field specifies the version number for the format of the authenticator. This field is of type ui_1.

authorization_data

The authorization_data field is used to pass authorization data from the principal on whose behalf a ticket was issued to the end service. If no authorization data is included, this field will be empty (i.e. it will have a length field indicating zero length). The data in this field are specific to the end service. It is expected that the field will contain the names of service specific objects, and the rights to those objects. This field is composed of a total length plus several subfields, each of type bytes_asn1. The total length, encoded in ASN.1 length format, includes the length of all the subfields and their length encodings (as for string arrays and host addresses). When the total length has been exhausted, there are no more subfields of authorization data. Although Kerberos is not concerned with the format of the contents of the subfields, it does carry type information (ad_type) in a subfield of type ui_2 immediately following each length subfield. The length of each authorization_data subfield includes the length of the data and the two bytes from the type subfield.



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

It is interesting to note that by specifying the authorization_data field of a proxy and leaving the host addresses blank, one is able to create a capability.

ad_type is a subfield of type ui_2 which specifies the format for the ad_data subfield. The meanings of the bits in the subfield are indicated below. Bit 0 is the most significant bit.

Bit(s)	*Name*	*Description*
0	RESERVED	Reserved for future expansion. Must be reset (0).
1	EXTERNAL	If this bit is reset (0), then the meaning of the ad_type field is defined in the Kerberos authorization proposal, and bits 2-15 encode a type from that proposal, with bit 2 as the most significant bit of an unsigned quantity. If this bit is set (1), then the meaning of the ad_type field is not defined in the Kerberos authorization proposal, and bits 3-15 are to be interpreted according to the value of bit 2 (REGISTERED).
2	REGISTERED	If this bit is set (1), the field type given by bits 3-15 is registered. If this bit is reset (0), then the field type is not registered, and the field type given by bits 3-15 has been arbitrarily chosen by the implementor, and are not guaranteed to be unique (They can be thought of as a ''magic number'').
3-15	FIELD-TYPE	These bits specify the field type or the unregistered magic number. They are to be interpreted as an unsigned integer, with bit 3 as the most significant bit.

An empty authorization data field (length zero in the total_length field) indicates that there is

no authorization data.

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 of type timestamp.

caddr This field in a ticket contains zero or more 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 to issue or accept zero-address tickets is a policy decision and is left to the Kerberos and end-service administrators. 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.

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.

This field is of type hostaddrs; its encoding is specified in section 5.1.8.

checksum_type
This field appears in the authenticator and the KRB_SAFE message, and specifies the algorithm used to generate the data checksum. A list of the pre-defined values for this field appears in section 5.2. This field is of type ui_2.

checksum This field appears in the authenticator and contains an optional checksum of the application data that is to follow. This field is of type bytes_asn1.

ckvno This field contains the client's key version number. It precedes the ciphertext in the KRB_AS_REP message, specifying which version of the client's secret key was used for the encrypted portion of the message. This field is of type ui_1.

cmsec This field contains the millisecond part of the client's timestamp. Its value (before encryption) ranges from 0 to 999. It often appears along with **ctime**. The two fields are used in conjunction to specify a reasonably accurate timestamp. This field is of type ui_2.

cname This field contains the name part of the client's identity. It is of type string array.

crealm This field contains the name of the realm in which the client is attempting to be authenticated. It is of type string.

ctime This field contains the current time on the client's workstation. It is of type timestamp.

confounder

This field contains random data and appears at the beginning of text encrypted in a principal's secret key. Its purpose is to make chosen- and known-plaintext attacks more difficult. It is important to note that the existence of this field does not prevent a verifiable plaintext attack. It just prevents the use of a precomputed ciphertext dictionary to find the corresponding plaintext. The efficacy of the confounder depends on the ability of the cryptosystem to propagate changes at the start of the encrypted plaintext through the remainder of the ciphertext. This field is of type ui_2. XXX longer? XXX

endtime This field contains the time after which the ticket will not be honored (its expiration time). Together with 'starttime', this field specifies the life of the ticket. 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. This field is of type timestamp.

error 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 interpretation of error codes. This field is of type ui_4.

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). It is of type string.

etype This field specifies the type of encryption being used to encrypt the ciphertext part of a ticket or message. A list of the pre-defined values for this field appears in section 5.2. This field is of type ui_2.

flags This field, of type flags, indicates which of various options were used or requested when the ticket was issued. It is a bit-field, where the selected options are indicated by the bit being set (1), and the unselected options and reserved fields being reset (0). Bit 0 is the most significant bit.

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 present ticket-granting ticket. This flag is reset by default, but users may request that it be set 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 (however, the end result can still be achieved if the user engages in the AS exchange from a remote host).
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. This flag is set by default. It allows proxies for specific services. For example, it allows a print server to access a client's files on a particular file server in order to satisfy a print request.
4	PROXY	When set, this flag indicates that a ticket is a proxy. It tells the end service that the client is acting on behalf of the principal, but may in fact be a different principal. A service might check this, and if a proxy, require additional authentication from the agent itself in order to provide an audit trail.
5	MAY-POSTDATE	The MAY-POSTDATE flag is normally only interpreted by the TGS, and can be ignored by end servers. This flag must be set in order to issue a postdated ticket based on the present ticket-granting ticket. It is reset by default. This flag does not allow one to obtain a postdated ticket-granting ticket. Post dated 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 of the ticket-granting ticket. The KDC may limit how far in the future a ticket may be postdated.
6	POSTDATED	This flag indicates that this ticket has been postdated. The end-service can check the 'authtime' field to see when the original authentication occurred. Some services may choose to reject post-dated tickets, or they may only accept them within a certain period after the original authentication.
7	INVALID	This flag indicates that a ticket is invalid. A postdated ticket will usually be issued in this form, and it must be validated by the KDC before it can be used, but after its 'starttime'. The validation is required so that postdated tickets which have been stolen before their 'starttime' can be rendered permanently invalid (through the hot-list mechanism).
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 new ticket that expires at a later date. This allows the life of a ticket to be extended without having to enter a password again, while providing some mechanism for cancellation of the right to use the ticket at renewal time. If the ticket is not renewed by its expiration time, then renewal will not be allowed. The RENEWABLE flag is reset by default. If set, then the 'renew_till' field contains a time after which the ticket may not be renewed.

9	INITIAL	This flag indicates that this ticket was issued using the initial request protocol. It was returned to the client encrypted in the client's secret key, and the request was not based on a ticket-granting ticket. Applications that want to require the entering of a password can check to see that this flag is set. An example of an application that would benefit from such a restriction is a password-changing program, which would traditionally require timely presentation of both old and new passwords.
10	DUPLICATE-SKEY	This flag indicates that the session key in this ticket may be used in other tickets as well. Other principals besides the named principal may know the session key. The ability to use the same session key in more than one ticket allows a key to be shared with more than one other principal. This is useful for implementing protocols in which all principals are trusted, and where information is broadcast to more than one other principal. Normal servers will not accept authentication based on a ticket that has this flag set (see the discussion of REUSE-SKEY under **kdc_options**, below).
11-31	RESERVED	Reserved for future use.

from This field is included in both the KRB_AS_REQ and KRB_TGS_REQ ticket requests. It specifies the desired start time for the requested ticket. Unless the request is for a postdated ticket, this field must be filled with zeros. This field is of type timestamp.

kdc_options

This field, of type flags, 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). Bit 0 is the most significant bit.

Bit(s)	*Name*	*Description*
0	RESERVED	Reserved for future expansion of this field.
1	FORWARDABLE	The FORWARDABLE option indicates that the ticket to be issued is to have its forwardable flag set. It may only be set on the initial request, or in a subsequent request if the ticket-granting ticket on which it is based is also forwardable.
2	FORWARDED	The FORWARDED option indicates that this is a request for forwarding. This option is only specified in a request to the ticket-granting server and will only be honored if the ticket-granting ticket on which it is based is forwardable. 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 on which it is based is proxiable. 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 'renew_till' field contains the desired absolute expiration time for the ticket.
9	UNUSED	This option is presently unused.
10	DUPLICATE-SKEY	The DUPLICATE-SKEY option indicates that the ticket to be issued is to have its DUPLICATE-SKEY flag set. This option may be requested at any time. This option does not duplicate the session key. Instead, it simply sets the flag in the ticket so that the session key can be reused at a later time.
11-26	RESERVED	Reserved for future use.
27	RENEWABLE-OK	The RENEWABLE-OK option indicates that a renewable ticket will be acceptable if a ticket with the requested life can not otherwise be provided. If a ticket with the requested life can not 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 second ticket granting ticket provided.

29	REUSE-SKEY	This option is used only by the ticket-granting service. The REUSE-SKEY option indicates that the session key to be assigned to the new ticket is to be taken from the second ticket provided. This option will only be honored if the second ticket has the DUPLICATE-SKEY flag set.
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 the renew_till field has not passed. (XXX Question: Should the ticket to be renewed be passed as a second ticket, or in the authenticator?).
31	VALIDATE	This option is used only by the ticket-granting service. The VALIDATE option indicates that the present request is to validate a post-dated 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 can not be validated before its start time. The ticket presented for validation is encrypted in the key of the server for which it is valid. (XXX Question: Should the ticket to be renewed be passed as a second ticket, or in the authenticator. Also, might it be better if invalid tickets were encrypted in the key for the ticket-granting server?)

keytype This field specifies the type of the session key included in the ticket. It will almost always correspond to the type of encryption specified by etype (it might not correspond, for example, if the etype uses an alternate checksum algorithm for an integrity check). A list of the pre-defined values for this field appears in section 5.2. This field is of type ui_2.

ktime This field contains the current time on the Kerberos server. It may be used (optionally) by an application to synchronize the clock of the client's workstation with that of the Kerberos server. This field is of type timestamp.

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.

This field is of type bytes_asn1. The contents must be a multiple of five (5) octets in length. Each five-octet portion (aligned with the start of the field contents) contains a one-octet **lr_type** subfield, followed by a ui_4 **lr_value** subfield. There may be several such sub-fields in a given last_req field. If the encoding indicates a zero (0) length, then there are no subfields or values to be examined.

lr_type This sub-field indicates the way that the following **lr_value** subfield is to be interpreted. Bit 0 is the most significant bit. The meanings of the bits are as follows:

Bit(s)	*Name*	*Description*
0	THIS-SERVER-ONLY	If set, the time refers to the responding server only. If reset, it applies to all servers for the realm.
1-7	INTERPRETATION	These bits are interpreted as an unsigned quantity, with bit 7 as the least significant bit. If this quantity is zero (0), then the lr_value subfield is the time of last initial request for a TGT. If it is one (1), then the lr_value subfield is the time of last initial request. If it is two (2), then the lr_value subfield is the time of issue for the newest ticket granting ticket used. If it is three (3), then the lr_value subfield is the time of the last renewal. If it is four (4), then the lr_value subfield is the time of last request (of any type).

msg_type This field indicates the type of a protocol message. It is of type ui_1.

pad This field fills the data in a message to a boundary specified by the cryptosystem in use. Some cryptosystems may use part of the pad to include an integrity checksum of the message.

key_exp This field specifies the time and date on which the principal's key in the Kerberos database expires. If imminent, the user should be warned. This field is of type timestamp.

pvno This field is included in each message, and specifies the protocol version number. This document specifies protocol version 5. This field is of type ui_1.

renew_till This field is included in tickets that are renewable. 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. This field is of type timestamp.

rtime This field is the requested renew_till time sent from a client to the KDC in a ticket request. It is optional. This field is of type timestamp.

second_ticket
A second ticket may be optionally included in a request to the ticket-granting server. If the SAME-SKEY option has been specified, then the second ticket contains the session key to be assigned to the new ticket. If the ENC-TKT-IN-SKEY option has been specified, then the session key from the second ticket will be used in place of the server's key to encrypt the new ticket. This field is of type bytes_asn1.

session This field contains the session key assigned by the KDC, to be used between the client and the server specified in the ticket. The type of this field is bytes_asn1.

skvno This field specifies the version number for the server's secret key. It is of type ui_1.

smsec This field contains the millisecond part of the server's timestamp. It's value ranges from 0 to 999. It appears along with **stime**. The two fields are used in conjunction to specify a reasonably accurate timestamp. This field is of type ui_2.

sname This field specifies the name part of the server's identity. It is of type string array.

srealm This field specifies the realm part of the server's identity. It also serves to identify the realm

that issued the ticket. This field is of type string.

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. This field is of type timestamp.

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

till This field contains the expiration date requested by the client in a ticket request. This field is of type timestamp.

tkt_vno This field specifies the version number for the ticket format. It is of type ui_1.

transited This field, of type bytes_asn1, indicates 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.

If a ticket is issued based on a ticket-granting ticket (TGT) issued by the local realm then the transited field should be passed through unchanged. When a ticket is issued based on a TGT issued by another realm then the name of the realm that issued the TGT should be added to the transited field. 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 from intentionally leaving out its own name.

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 in a manner that is optimized for the normal case of inter-realm communication.

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.

Realm names in the transited field are separated by a ",". A realm name ending with a "." is interpreted as being prepended to the previous realm. For example, we can encode traversal of EDU, MIT.EDU, ATHENA.MIT.EDU, WASHINGTON.EDU, and CS.WASHINGTON.EDU as:

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

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

"EDU,MIT.,WASHINGTON.EDU"

A null subfield preceding or following a "," indicates that all realms between the previous realm and the next realm have been traversed. Thus, "," means that the whole tree has been traversed, but ",MIT.EDU,WASHINGTON.EDU," means that everything up to MIT.EDU, and everything below WASHINGTON.EDU (inclusive) have been traversed, but everything between them has been bypassed.

DRAFT 2

7. Message Specifications

The following sections describe the exact contents and encoding of protocol messages and objects. Descriptions of the individual fields are described above in section 6.

7.1. Tickets and Authenticators

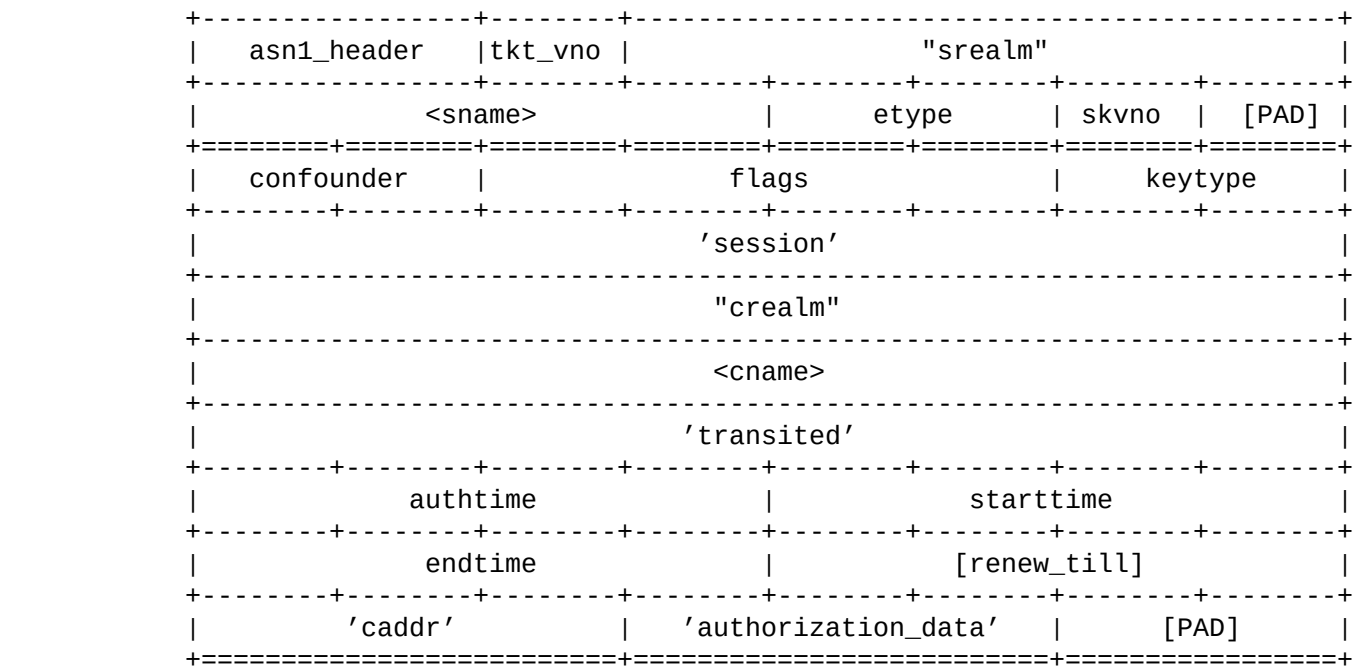
This section describes the format and encryption parameters for tickets and authenticators. When a ticket or authenticator is included in a protocol message it is treated as an opaque object. The length can be determined from the ASN.1 header that appears at its start.

7.1.1. Tickets

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

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	tkt_vno	ticket format version number (= 5)
<= 128 octets	string	srealm	service's realm
<= 128 octets	stringarray	sname	service's name
2 octets	ui_2	etype	encryption type
1 octet	ui_1	skvno	service key version number
variable	PAD	pad	null pad to blocksize-octet multiple
=======			
2 octets	confounder	confounder	random data
4 octets	flags	flags	bit field of flags
2 octets	ui_2	keytype	encryption key type of session key
variable	bytes_asn1	session	session key
<= 128 octets	string	crealm	client's realm
<= 128 octets	string	<cname>	client's name
<= 256 octets	bytes_asn1	transited	list of transited realms
4 octets	timestamp	authtime	time of client's initial authentication
4 octets	timestamp	starttime	beginning of valid period for this ticket
4 octets	timestamp	endtime	end of valid period
4 octets	timestamp	renew_till	OPTIONAL: end of renewable life
<= 256 octets	hostaddr	caddr	client's host address(es)
<= 512 octets	bytes_asn1	authorization_data	client-supplied authorization data (possibly empty)
variable	PAD	pad	null pad to blocksize-octet multiple
=======			

The data between double dashed lines above are encrypted in the key shared by Kerberos and the end server (the server's secret key).

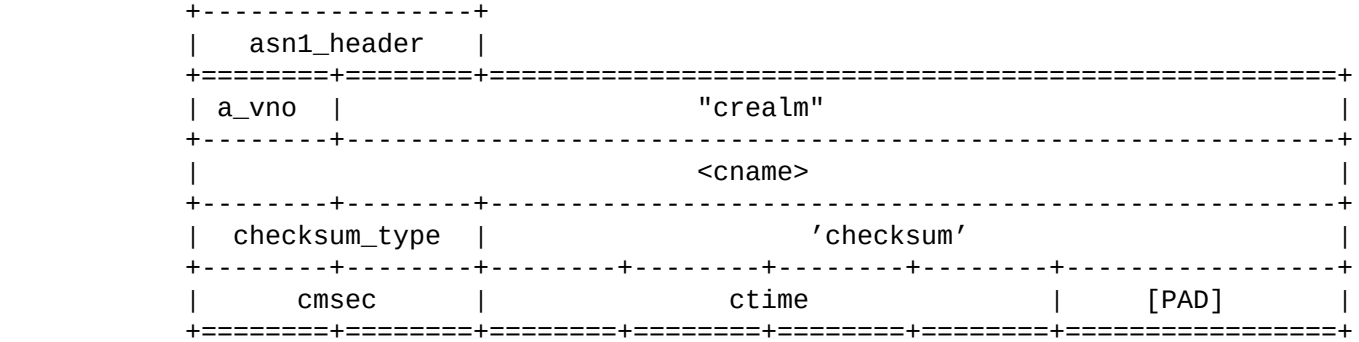


The optional **renew_till** field is only present if the RENEWABLE flag is set in the **flags** field.

7.1.2. 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 and to help the server detect replays. An authenticator contains the following fields. Those surrounded by double dashes are encrypted in the session key shared by the client and the server:

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	authenticator_vno	authenticator format version number (= 5)
<= 128 octets	string	crealm	client's realm
<= 128 octets	stringarray	cname	client's name
2 octets	ui_2	checksum_type	Type of application specific checksum
variable	bytes_asn1	checksum	Application specific checksum
2 octets	ui_2	cmsec	client timestamp (millisecond portion)
4 octets	timestamp	ctime	timestamp in seconds
variable	PAD	pad	null pad to blocksize-octet multiple



7.2. Authentication Server (AS) message specifications

This section specifies the format of the messages used in the exchange with the authentication service. The format of error messages appears in section 7.7.

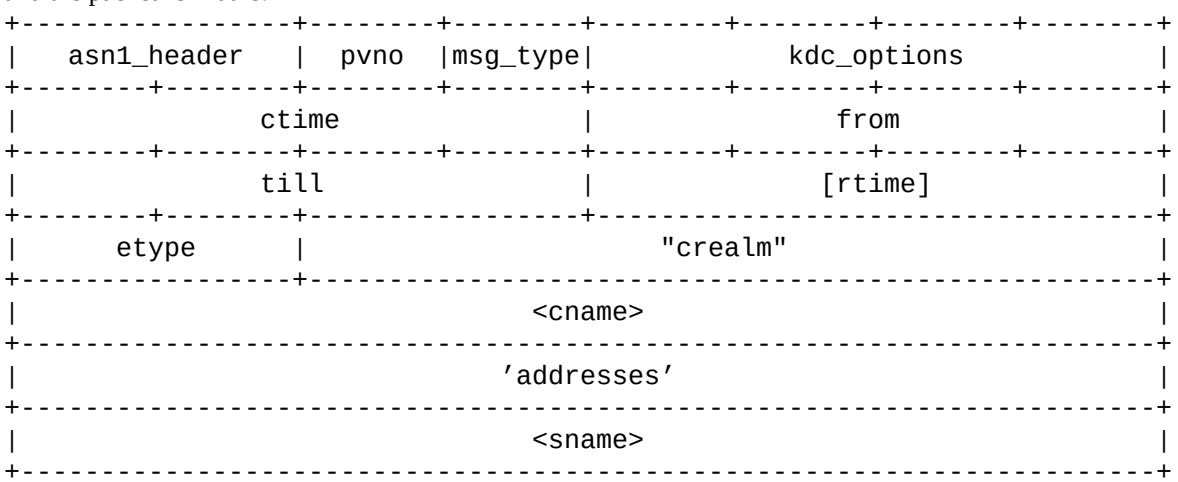
7.2.1. KRB_AS_REQ definition

The KRB_AS_REQ message (sent from the client to the Authentication Server) contains the Kerberos protocol version number, the KRB_AS_REQ message type, the desired options, the identity of the client for which the credentials are requested, the host addresses to be included in the ticket, the desired start and end times of the ticket life, the identity of the server to which the credentials will be presented, and the local host's timestamp.

The message fields are:

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protocol version number (= 5)
1 octet	type	msg_type	message type (= KRB_AS_REQ)
4 octets	flags	kdc_options	options desired
4 octets	timestamp	ctime	client's timestamp in seconds
4 octets	timestamp	from	desired start time
4 octets	timestamp	till	desired expiration time
4 octets	timestamp	rtime	OPTIONAL: desired renew_till
2 octets	ui_2	etype	desired encryption type for reply
<= 128 octets	string	crealm	client's realm
<= 128 octets	stringarray	cname	client's name
<= 256 octets	hostaddr	addresses	host address(es) for ticket
<= 128 octets	stringarray	sname	service's name

and the packet format is:



7.2.2. KRB_AS_REP definition

The KRB_AS_REP message is an instance of the KRB_KDC_REP message with the message type set to KRB_AS_REP, and where the ciphertext portion is encrypted in the client's secret key.

7.2.3. KRB_KDC_REP definition

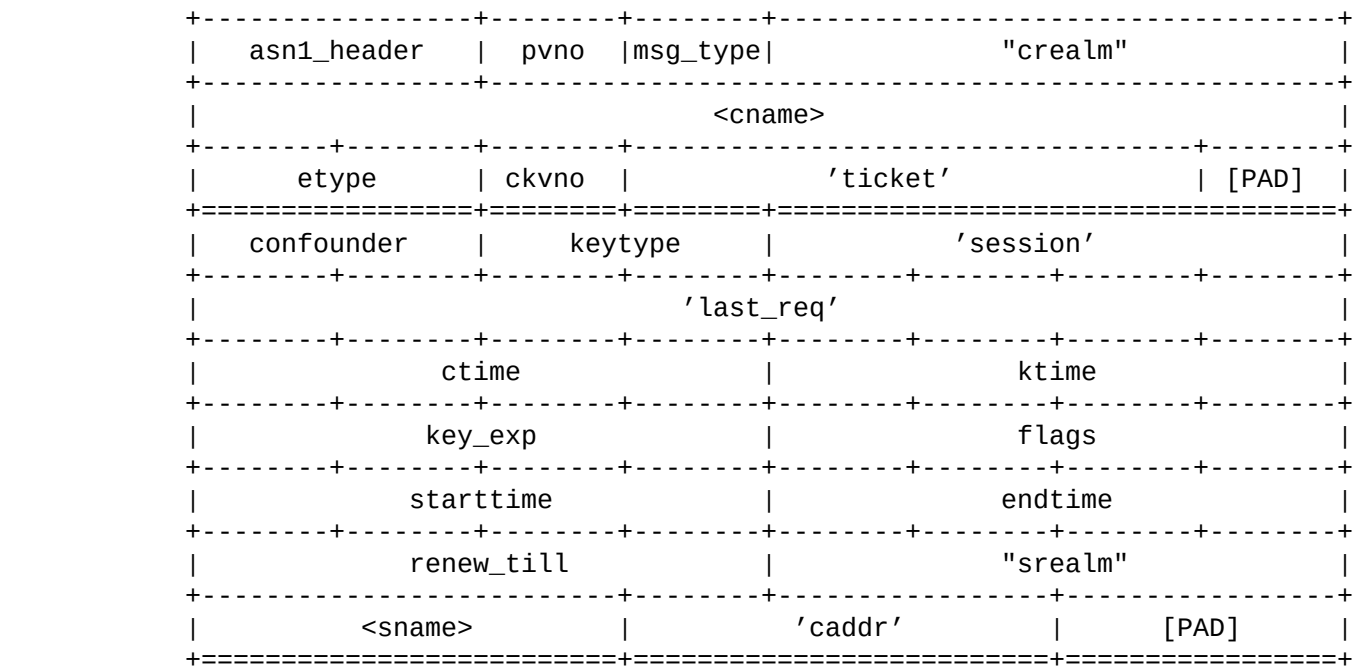
The KRB_KDC_REP message format is used for the reply from the KDC for either an initial (AS) request, or a subsequent (TGS) request. There is no message type for KRB_KDC_REP. Instead, the type will be one of KRB_AS_REP, or KRB_TGS_REP. The key used to encrypt the ciphertext part of the reply

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 ckvno. For KRB_TGS_REP, the ciphertext is encrypted in the session key from the ticket granting ticket used in the request. In that case, ckvno will be zero.

The KRB_KDC_REP message contains the following fields:

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protocol version number (= 5)
1 octet	type	msg_type	message type (either KRB_AS_REP or KRB_TGS_REP)
<= 128 octets	string	crealm	client's realm
<= 128 octets	stringarray	cname	client's name
2 octets	ui_2	etype	encryption type
1 octet	ui_1	ckvno	client's key version number
variable	ticket	ticket	ticket for the service
variable	PAD	pad	null pad to blocksize-octet multiple
=======			
2 octets	confounder	confounder	random data
2 octets	ui_2	keytype	encryption key type of session key
variable	bytes_asn1	session	session key
<= 128 octets	bytes_asn1	last_req	last request information
4 octets	timestamp	ctime	client's timestamp (used as nonce)
4 octets	timestamp	ktime	KDC timestamp (for sync)
4 octets	timestamp	key_exp	principal expiration date
4 octets	flags	flags	flags set in ticket
4 octets	timestamp	starttime	ticket start date
4 octets	timestamp	endtime	ticket expire date
4 octets	timestamp	renew_till	OPTIONAL: end of renewable_life
<= 128 octets	string	srealm	server's realm
<= 128 octets	stringarray	sname	server's name (to link ticket and ciphertext)
<= 256 octets	hostaddr	caddr	client's host address(es)
variable	PAD	pad	null pad to blocksize-octet multiple
=======			

in the following format:



The ticket should be thought of as an opaque object. It is of type bytes_asn1, and its first few octets encode its length. Although the ticket itself is a multiple of blocksize octets, the ticket field is not (because of the length encoding). It is not necessary for the ticket to be aligned.

The encrypted part of the response (shown above between double dashed lines) must begin and end on a blocksize boundary. Encryption occurs under the client's secret key if this is a message of type KRB_AP_REP. If the type is KRB_TGS_REP, then the session key from the ticket-granting ticket is used for the encryption.

The caddr field will contain the requested addresses (for modification detection) if the message is of type KRB_AS_REP. If the type is KRB_TGS_REP, then this field will only be filled in if the request was for a proxy or forwarded ticket. If not, then the addresses contained in the ticket are the same as included in the ticket-granting ticket.

7.3. Client/Server (CS) message specifications

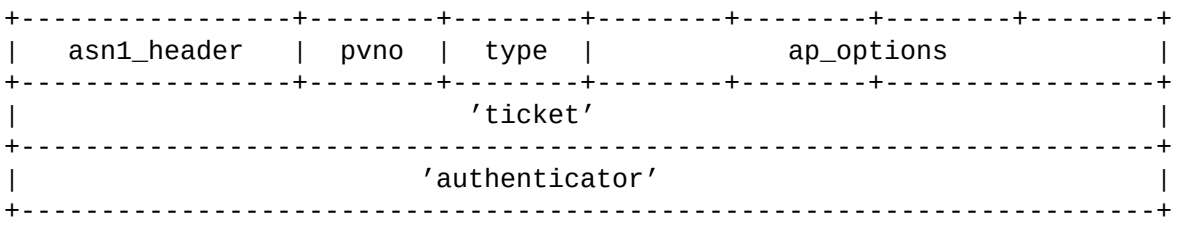
This section specifies the format of the messages used for the authentication of the client to the end server.

7.3.1. 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 ticket and authenticator are included in fields of type bytes_asn1, and the lengths are encoded in the initial octets. The KRB_AP_REQ message is often referred to as the "authentication header".

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protocol version number (= 5)
1 octet	type	type	message type (= KRB_AP_REQ)
4 octets	flags	ap_options	message options
variable	ticket	ticket	Ticket
variable	ticket	authenticator	Authenticator

The message format is:

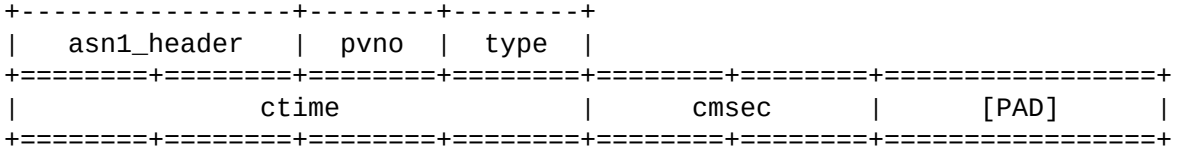


7.3.2. KRB_AP_REP definition

The KRB_AP_REP message contains the Kerberos protocol version number, the message type, and an encrypted timestamp. The message is sent in response to a request for mutual authentication.

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protocol version number (= 5)
1 octet	type	type	message type (= KRB_AP_REP)
========			
4 octets	timestamp	ctime	ctime from authenticator (nonce)
2 octets	ui_2	cmsec	cmsec from authenticator
variable	PAD	PAD	null pad to blocksize-octet multiple
=======			

The data between the double dashed lines are encrypted in the shared session key.



7.3.3. Error message reply

If an error occurs, the KRB_ERROR message will be sent in response. The "cname" field may be an empty string array and the "crealm" field may be an empty string if the server cannot determine their appropriate values from the corresponding KRB_AP_REQ message. The ctime and cmsec fields will contain the values read from the authenticator if they were successfully read.

7.4. Ticket-granting service (TGS) message definition

This section specifies the format of the messages used to request additional ticket from the ticket granting server after the initial ticket granting ticket has been received.

7.4.1. KRB_TGS_REQ definition

The KRB_TGS_REQ message consists of an authentication header (KRB_AP_REQ, see section 7.3), and fields containing information about the specific request. These fields include the desired options for the new ticket, the host addresses to insert in the ticket, the desired start and expiration times, the name of the server for which credentials are to be obtained, and the client's timestamp. The client may optionally include addresses from which the new ticket is to be valid, a second ticket, or a free-form sequence of bytes (the authorization_dat field) to be sealed in the ticket and used to assist in authorization decisions by the server.

Length	*Type*	*Label*	*Value*

DRAFT 2

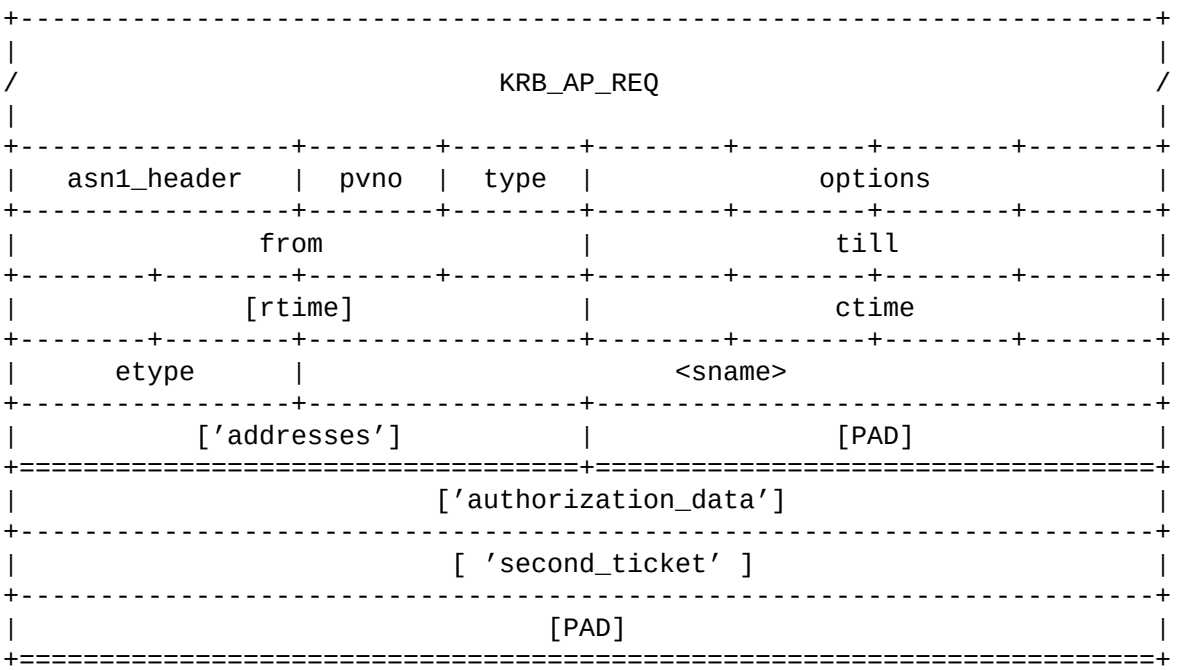
variable	KRB_AP_REQ	KRB_AP_REQ	KRB_AP_REQ header
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protovol version number (= 5)
1 octet	type	type	message type (= KRB_TGS_REQ)
4 octets	flags	options	options desired
4 octets	timestamp	from	desired start time
4 octets	timestamp	till	desired expiration time
4 octets	timestamp	rtime	OPTIONAL: desired renew_till
4 octets	timestamp	ctime	client's timestamp in seconds
2 octets	ui_2	etype	desired encryption type for reply
<= 128 octets	stringarray	sname	name of service
<= 256 octets	hostaddr	addresses	OPTIONAL: host address(es) for ticket
variable	PAD	PAD	null pad to blocksize-octet multiple
========			
<= 512 octets	bytes_asn1	authorization_dat	OPTIONAL: authorization data
variable	ticket	second_ticket	OPTIONAL: additional ticket
variable	PAD	PAD	null pad to blocksize-octet multiple
========			

The data between dashed lines are encrypted in the session key from the ticket granting ticket. The optional fields are only included if necessary to perform the operation specified in the "options" field. If none of the three optional fields are included, then the encrypted part of the request is eliminated altogether. The optional fields are followed by a PAD.

The user-supplied checksum of the KRB_AP_REQ header of the KRB_TGS_REQ message is a checksum of the KRB_TGS_REQ fields (from pvno to second_ticket, inclusive) before encryption. This checksum enables the KDC to determine whether the encrypted portions of the KRB_TGS_REQ message were modified in transit.

It should be noted that in KRB_TGS_REQ, the protocol version number appears twice, and two different message types appear. The authentication header (KRB_AP_REQ) includes these fields, as does the KRB_TGS_REQ message itself.

The packet format is (optional fields in [brackets]):



7.4.2. KRB_TGS_REP definition

The KRB_TGS_REP is an instance of the KRB_KDC_REP message described in section 7.2.3 with the message type KRB_TGS_REP, and where the ciphertext portion is encrypted in the session key from the ticket granting ticket. The 'caddr' field in the KRB_TGS_REP is set to the contents of the 'addresses' field of the corresponding KRB_TGS_REP (if present) or the 'caddr' field of the TGT in the KRB_TGS_REP (if no 'addresses' field is supplied).

7.5. 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 message).

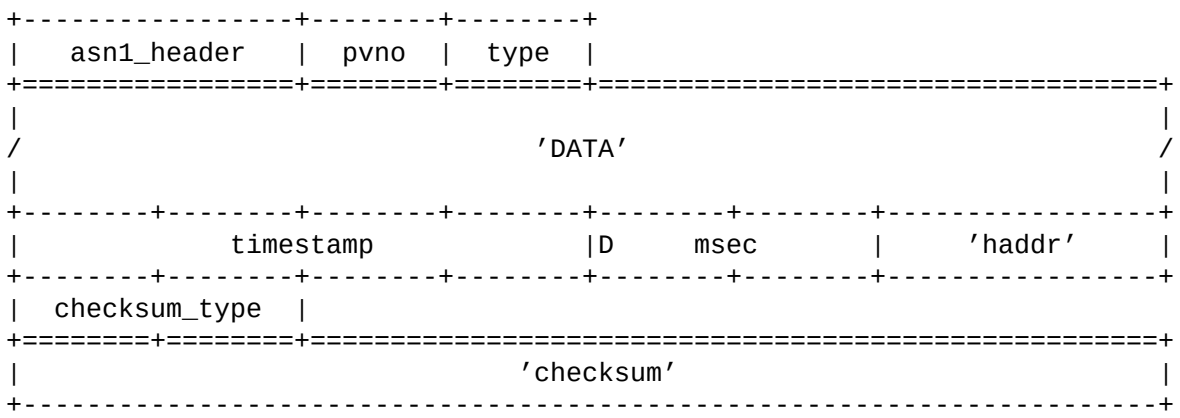
7.5.1. KRB_SAFE definition

The KRB_SAFE message contains user data along with a cryptographic checksum based on the session key. The message fields are:

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protocol version number (= 5)
1 octet	type	type	message type (= KRB_SAFE)
========			
variable	bytes_asn1	DATA	user data
4 octets	timestamp	timestamp	message sender's timestamp
2 octets	ui_2	msec	sender's timestamp (millisecond portion)
1 bit	--	D	direction in most significant bit of msec field
<= 256 octets	hostaddr	haddr	sender's host address(es)
2 octets	ui_2	checksum_type	type of checksum

variable	bytes_asn1	checksum	cryptographic checksum

The data between the dashed lines above are computed into the checksum. The packet format is:



7.6. 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 message).

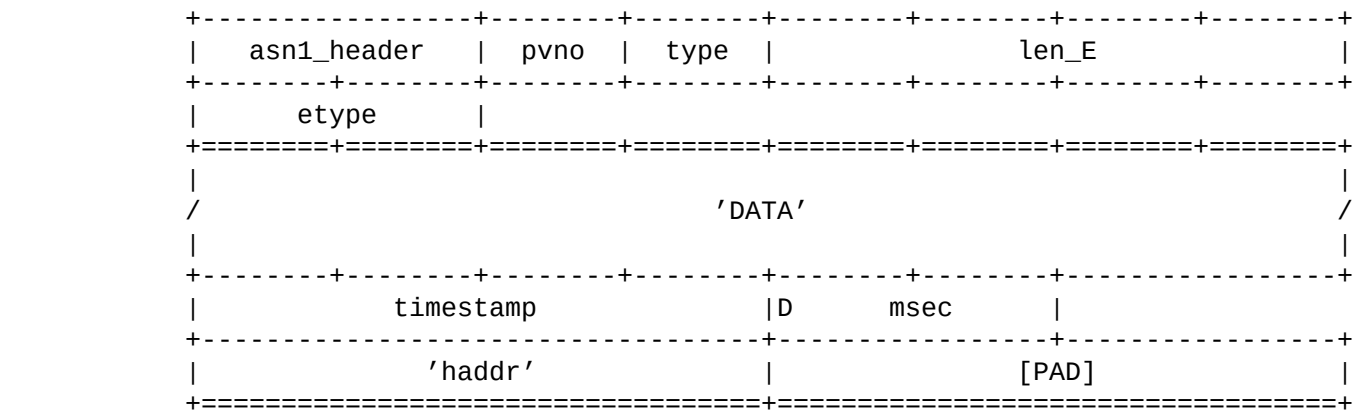
7.6.1. KRB_PRIV definition

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

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protocol version number (= 5)
1 octet	type	type	message type (= KRB_PRIV)
4 octets	ui_4	len_E	length of encrypted portion
2 octets	ui_2	etype	encryption type
variable	bytes_asn1	DATA	user data
4 octets	timestamp	timestamp	sender's timestamp (seconds)
2 octets	ui_2	msec	sender's timestamp (millisecond portion)
1 bit	--	D	direction in most significant bit of msec field
<= 256 octets	hostaddr	haddr	sender's host address(es)
variable	PAD	PAD	null pad to blocksize-octet multiple

The fields between the double dashed lines are encrypted in the session key before transmission.

The packet format is:



7.7. Error message specification

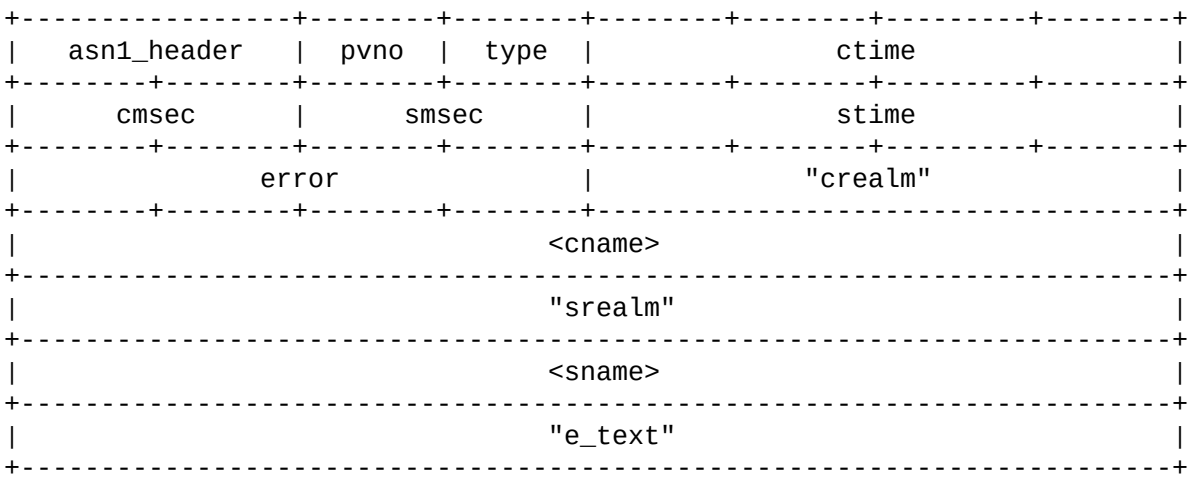
This section specifies the format for the KRB_ERROR message. The fields included in the message are intended to return as much information as possible about an error. It is not expected that all the information required by the fields will be available for all types of errors. If information is not available, the corresponding field will be filled with zeroes (if it is numeric), or be a zero-length string (if it is a string). To interpret the error code, see section 8.

7.7.1. KRB_ERROR definition

The KRB_ERROR message consists of the following fields:

Length	*Type*	*Label*	*Value*
variable		asn1_header	ASN.1 compatibility header
1 octet	ui_1	pvno	protocol version number (= 5)
1 octet	type	type	message type (= KRB_ERROR)
4 octets	timestamp	ctime	client’s timestamp in seconds
2 octets	ui_2	cmsec	client’s timestamp (millisecond portion)
2 octets	ui_2	smsec	server’s timestamp (millisecond portion)
4 octets	timestamp	stime	server’s timestamp in seconds
4 octets	ui_4	error	error code
<= 128 octets	string	crealm	client’s realm
<= 128 octets	stringarray	cname	client’s name
<= 128 octets	string	srealm	server’s realm
<= 128 octets	stringarray	sname	server’s name
<= 128 octets	string	e_text	additional error text

in the following format:



8. Constants

The following table lists the constants used in the protocol and defines their meanings.

Label	*Value*	*Meaning or MIT code*
pvno	5	current Kerberos protocol version number
message types		
KRB_AS_REQ	2	Request for initial authentication
KRB_AS_REP	4	Response to KRB_AS_REQ request
KRB_AP_REQ	6	application request to server
KRB_AP_REQ_MUTUAL	8	KRB_AP_REQ with request for mutual authentication
KRB_AP_REP	10	Response to KRB_AP_REQ_MUTUAL
KRB_TGS_REP	12	Response to KRB_TGS_REQ request
KRB_SAFE	14	Safe (checksummed) application message
KRB_PRIV	12	Private (encrypted) application message
KRB_ERROR	32	Error response
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 entries for principal in Kerberos 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 accomodate requested option
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_ETYPE_NOSUPP	46	No support for encryption type
KRB_AP_ERR_MUT_FAIL	47	Mutual authentication failed
KRB_ERR_FIELD_TOOLONG	50	Field is too long for this implementation

9. REFERENCES

1. 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).
2. J. G. Steiner, B. C. Neuman, and J. I. Schiller, ‘‘Kerberos: An Authentication Service for Open Network Systems,’’ pp. 191-202 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).
3. R. M. Needham and M. D. Schroeder, ‘‘Using Encryption for Authentication in Large Networks of Computers,’’ *Communications of the ACM* **21**(12), pp. 993-999 (December, 1978).
4. Dorothy E. Denning and Giovanni Maria Sacco, ‘‘Timestamps in Key Distribution Protocols,’’ *Communications of the ACM* **24**(8), pp. 533-536 (August 1981).
5. Don Davis and Ralph Swick, *Workstation Services and Kerberos Authentication at Project Athena*, MIT Project Athena (March 3, 1989).
6. National Bureau of Standards, ‘‘Data Encryption Standard,’’ Federal Information Processing Standards Publication 46, Washington, D.C. (1977).
7. National Bureau of Standards, ‘‘DES Modes of Operation,’’ Federal Information Processing Standards Publication 81, Springfield, VA (1980).
8. P. J. Levine, M. R. Gretzinger, J. M. Diaz, W. E. Sommerfeld, and K. Raeburn, *Section E.1: Service Management System*, M.I.T. Project Athena, Cambridge, Massachusetts (1987).
9. J. L. Smith, ‘‘The design of Lucifer, a cryptographic device for data communications.,’’ RC 3326, IBM T.J. Watson Research Center, Yorktown Heights, NY (April 15, 1971).
10. International Organization for Standardization, ‘‘ISO Information Processing Systems - Data Communication - High-Level Data Link Control Procedure - Frame Structure,’’ 3309, ISO (October 1984). 3rd Edition.
11. Ralph C. Merkle, *A Fast Software One Way Hash Function*, Xerox PARC, Palo Alto, CA (in preparation).

A. Pseudo-code for protocol processing

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

A.1. KRB_AS_REQ generation

```
req.asn1_header = HEADER; /* constant except for length encoding */
req.pvno = 5;
req.type = KRB_AS_REQ;
req.kdc_options = (set according to user's preferences);
req.cname = name;       /* passed in by user */
req.crealm = realm;     /* passed in by user */
req.addresses = (host-address);
req.from = 0; /* unless user specifies a specific start time */
req.till = 0; /* unless user specifies a specific end time */
if renewable then
        /* user wants renewable */
        req.rtime = (time specified by user);
endif
req.sname = (service-name) /* usually "krbtgt",  "localrealm" */
get system_time;
req.ctime = system_time.seconds;

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.2. KRB_AS_REQ verification and KRB_AS_REP generation

```
parse request 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 */
        return KRB_ERROR message with
                code == KDC_ERR_C_PRINCIPAL_UNKNOWN;
endif
if (!server) then
        /* no server in Database */
        return KRB_ERROR message with
                code == KDC_ERR_S_PRINCIPAL_UNKNOWN;
endif

session = generate_random_session_key();

tkt.asn1_header = HEADER; /* constant except for length encoding */
tkt.vno = 5;
tkt.sname = req.sname;
tkt.srealm = req.realm;
tkt.etype = (encryption-type); /* might be DES */
tkt.skvno = server.kvno;

pad(to cryptosystem boundary);

tkt.confounder = random();
tkt.flags = 0;

/* 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) then
        set(tkt.flags.FORWARDABLE);
endif
if (req.kdc_options.FORWARDED) then
        return KRB_ERROR, code KDC_ERR_BADOPTION;
endif
if (req.kdc_options.PROXIABLE) then
        set(tkt.flags.PROXIABLE);
endif
```

```
if (req.kdc_options.PROXY) then
        return KRB_ERROR, code KDC_ERR_BADOPTION;
endif
if (req.kdc_options.ALLOW-POSTDATE) then
        set(tkt.flags.ALLOW-POSTDATE);
endif

if (req.kdc_options.DUPLICATE-SKEY) then
        set(tkt.flags.DUPLICATE-SKEY);
endif
if (req.kdc_options.RENEW or req.kdc_options.VALIDATE or
   req.kdc_options.REUSE-SKEY or
   req.kdc_options.ENC-TKT-IN-SKEY) then
        return KRB_ERROR, code KDC_ERR_BADOPTION;
endif

tkt.keytype = (encryption-type); /* Presently DES */
tkt.session = session;
tkt.cname = req.cname;
tkt.crealm = req.crealm;
tkt.transited = "";

tkt.authtime = kdc_time;

if (req.kdc_options.POSTDATED) then
   set(tkt.flags.INVALID);
   if (against_postdate_policy(req.from)) then
        return KRB_ERROR, code KDC_ERR_POLICY;
   endif
   tkt.starttime = req.from;
else
        tkt.starttime = kdc_time;
endif
if (req.till = 0) then
        till = infinity;
else
        till = req.till;
endif

tkt.endtime = min(till,tkt.starttime+client.max_life,
                  tkt.starttime+server.max_life,
                        tkt.starttime+max_life_for_realm);

if (req.kdc_options.RENEWABLE-OK and (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) then
        set(tkt.flags.RENEWABLE);
        tkt.renew_till = min(rtime,starttime+client.max_rlife,
                             tkt.starttime+server.max_rlife,
                             tkt.starttime+max_rlife_for_realm);
else
        tkt.renew_till = OMIT; /* leave the renew_till field out */
endif

tkt.caddr = req.addresses;
tkt.authorization_data = "";

encrypt(appropriate part of ticket);

/* Start processing the response */

resp.asn1_header = HEADER; /* constant except for length encoding */
resp.pvno = 5;
resp.type = KRB_AS_REP;
resp.cname = req.cname;
resp.crealm = req.realm;
resp.etype = (encryption-type); /* Presently DES */
resp.ckvno = client.kvno;
resp.ticket = ticket;

pad(to cryptosystem boundary);

resp.confounder = random();
resp.keytupe = (encryption-type); /* Presently DES */

resp.session = session;
resp.ctime = req.ctime;
resp.ktime = kdc_time;

resp.last_req = fetch_last_request_info(client);

resp.princ_exp = client.expiration;
resp.flags = tkt.flags;
resp.sname = tkt.sname;
resp.srealm = tkt.srealm;

resp.starttime = tkt.starttime;
resp.endtime = tkt.endtime;

if (tkt.flags.RENEWABLE) then
        resp.renew_till = tkt.renew_till;
endif

resp.caddr = tkt.caddr;

pad(to cryptosystem boundary);

encrypt(appropriate part of response);
```

```
send(resp);
```

A.3. KRB_AS_REP verification

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

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

prompt_user_for(key);
decrypt(resp,key);
zero(key);

if (!integrity_ok(resp)) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if (req.cname != resp.cname) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if (req.realm != resp.crealm) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if (req.sname != resp.sname) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if (req.realm != resp.srealm) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if (req.ctime != resp.ctime) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if (req.addresses != resp.caddr) then
  destroy session key in resp;
        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 session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif

if ((req.from = 0) and
    (resp.starttime is not within allowable skew)) then
```

```
  destroy session key in resp;
        return KRB_AP_ERR_SKEW;
endif
if ((req.from != 0) and (req.from != resp.starttime)) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if ((req.till != 0) and (resp.endtime > req.till)) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if ((req.kdc_options.RENEWABLE) and
    (req.rtime != 0) and (resp.renew_till > req.rtime)) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif
if ((req.kdc_options.RENEWABLE-OK) and
    (resp.flags.RENEWABLE) and
    (req.till != 0) and
    (resp.renew_till > req.till)) then
  destroy session key in resp;
        return KRB_AP_ERR_MODIFIED;
endif

if near(resp.princ_exp) then
        print(warning message);
endif
save_for_later(ticket,session,client,server,times,flags);
```

A.4. KRB_TGS_REQ generation

```
/* Note that make_application_request might have to recursivly     */
/* call this routine to get the appropriate ticket granting ticket */

req.ahdr = make_application_request(krbtgt,srealm);

req.asn1_header = HEADER; /* constant except for length encoding */
req.pvno = 5;
req.type = KRB_TGS_REQ;
req.kdc_options = (set according to user's preferences);

req.from = 0; /* unless this is a request for a postdated ticket */
req.till = 0; /* unless user specifies a specific life */

if (renewable) then
        req.rtime = (time specified by user);
endif
req.sname = (the name of the desired service);
get system_time;
req.ctime = system_time.seconds;

pad(to cryptosystem boundary);

req.addresses = 0; /* Unless we are changing them */
req.authorization_dat = (as set by the user, null by default);
```

```
req.second_ticket = (second ticket if needed, null by default);

pad(to cryptosystem boundary);

encrypt(appropriate part of request);

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.5. 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. */

read_application_request(req);

/* 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. */

realm = realm_of_tgt(req.auth_hdr.ticket);

parse remainder of request;

server = lookup(req.sname,realm);

if (!server) then
        /* no server in Database */
        return KRB_ERROR message with
                code == KDC_ERR_S_PRINCIPAL_UNKNOWN;
endif

if (req.kdc_options.REUSE-SKEY) then
        decrypt(req.second_ticket);
        if (!req.second_ticket.flags.DUPLICATE-SKEY) then
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        session = req.second_ticket.session;
else
        session = generate_random_session_key();
endif

tkt.asn1_header = HEADER; /* constant except for length encoding */
tkt.vno = 5;

tkt.sname = req.sname;
```

```
tkt.srealm = realm;

tkt.etype = (encryption-type); /* Presently DES */
tkt.skvno = server.kvno;

pad(to cryptosystem boundary);

tkt.confounder = random();

tkt.flags = 0;
tkt.starttime = 0;

/* 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         */

tkt.caddr = req.auth_hdr.ticket.caddr;
resp.caddr = NULL; /* We only include this if they change */
if (req.kdc_options.FORWARDABLE) then
        if (!req.auth_hdr.ticket.flags.FORWARDABLE) then
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        set(tkt.flags.FORWARDABLE);
endif
if (req.kdc_options.FORWARDED) then
        if (!req.auth_hdr.ticket.flags.FORWARDABLE)
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        set(tkt.flags.FORWARDED);
        tkt.caddr = req.addresses;
        resp.caddr = req.addresses;
endif

if (req.kdc_options.PROXIABLE) then
        if (!req.auth_hdr.ticket.flags.PROXIABLE)
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        set(tkt.flags.PROXIABLE);
endif
if (req.kdc_options.PROXY) then
        if (!req.auth_hdr.ticket.flags.PROXIABLE)
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        set(tkt.flags.PROXY);
        tkt.caddr = req.addresses;
        resp.caddr = req.addresses;
endif

if (req.kdc_options.POSTDATE) then
        if (!req.auth_hdr.ticket.flags.POSTDATE)
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        set(tkt.flags.POSTDATE);
endif
```

```
if (req.kdc_options.POSTDATED) then
        if (!req.auth_hdr.ticket.flags.POSTDATE) then
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        set(tkt.flags.POSTDATED);
        set(tkt.flags.INVALID);
        if (against_postdate_policy(req.from)) then
                return KRB_ERROR, code KDC_ERR_POLICY;
        endif
        tkt.starttime = req.from;
  endif

if ((req.kdc_options.DUPLICATE-SKEY) or
    (req.kdc_options.REUSE-SKEY)) then
        set(tkt.flags.DUPLICATE-SKEY);
endif

if (req.kdc_options.VALIDATE) then
        if (!req.auth_hdr.ticket.flags.INVALID) then
                return KRB_ERROR, code KDC_ERR_POLICY;
        endif
        if (req.auth_hdr.ticket.starttime > kdc_time) then
                return KRB_ERROR, code KRB_AP_ERR_NYV;
        endif
        if (check_hot_list(req.auth_hdr.ticket)) then
                return KRB_ERROR, code KRB_AP_ERR_REPLAY;
        endif
        tkt = req.auth_hdr.ticket;
        clear(tkt.flags.INVALID);
endif

if (req.kdc_options.(any flag except ENC-TKT-IN-SKEY, RENEW,
                     and those already processed) then
        return KRB_ERROR, code KDC_ERR_BADOPTION;
endif

tkt.authtime = req.auth_hdr.ticket.authtime;

if (req.kdc_options.RENEW) then
  /* Note that if the endtime has already passed, the ticket would  */
  /* have been rejected in the initial authewntication stage, so    */
  /* there is no need to check again here                           */
        if (!req.auth_hdr.ticket.flags.RENEWABLE) then
                return KRB_ERROR, code KDC_ERR_BADOPTION;
        endif
        if (!req.auth_hdr.ticket.renew_till < kdc_time) then
                return KRB_ERROR, code KRB_AP_ERR_TKT_EXPIRED;
        endif
        tkt = req.auth_hdr.ticket;
        tkt.starttime = kdc_time;
        old_life = req.auth_hdr.ticket.endttime -
                   req.auth_hdr.ticket.starttime;
        tkt.endtime = min(req.auth_hdr.ticket.renew_till,
                          tkt.starttime + old_life);
```

```
else
        tkt.starttime = kdc_time;
        if (req.till = 0) then
                till = infinity;
        else
                till = req.till;
        endif
        tkt.endtime = min(till,tkt.starttime+client.max_life,
                          tkt.starttime+server.max_life,
                          tkt.starttime+max_life_for_realm,
                          req.auth_hdr.ticket.endtime);

        if (req.kdc_options.RENEWABLE-OK and
            (tkt.endtime < req.till) and
            req.auth_hdr.ticket.flags.RENEWABLE) then
                /* we set the RENEWABLE option for later processing */
                set(req.kdc_options.RENEWABLE);
                req.rtime = min(req.till,
                                req.auth_hdr.ticket.renew_till);
        endif
endif

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

if (req.kdc_options.RENEWABLE and
    req.auth_hdr.ticket.flags.RENEWABLE) then
        set(tkt.flags.RENEWABLE);
        tkt.renew_till = min(rtime,starttime+client.max_rlife,
                             tkt.starttime+server.max_rlife,
                             tkt.starttime+max_rlife_for_realm,
                             tkt.auth_hdr.ticket.renew_till);
else
        tkt.renew_till = OMIT; /* leave the renew_till field out */
endif
tkt.authorization_data = req.auth_hdr.ticket.authorization_data +
                         req.authorization_data;

tkt.keytype = (encryption-type); /* Presently DES */
tkt.session = session;
tkt.cname = req.auth_hdr.ticket.cname;
tkt.crealm = req.auth_hdr.ticket.crealm;

if (realm_of_tgt(req.auth_hdr.ticket) = req.auth_hdr.ticket.realm) then
        /* tgt issued by local realm */
        tkt.transited = req.auth_hdr.ticket.transited.
else
        tkt.transited =
        compress_transited(req.authenication_header.ticket.transited +
                           req.auth_hdr.ticket.realm)
endif
```

```
if (req.kdc_options.ENC-TKT-IN-SKEY) then
        decrypt(req.second_ticket);
        encrypt(appropriate part of ticket,req.second_ticket.session);
else
        encrypt(appropriate part of ticket,server.key);
endif

resp.asn1_header = HEADER; /* constant except for length encoding */
resp.pvno = 5;
resp.type = KRB_TGS_REP;
resp.cname = req.auth_hdr.ticket.cname;
resp.crealm = req.auth_hdr.ticket.crealm;
resp.etype = (encryption-type); /* Presently DES */

resp.ckvno = 0; /* We are using the session key */
resp.ticket = ticket;

pad(to cryptosystem boundary);

resp.confounder = random();
resp.keytupe = (encryption-type); /* Presently DES */
resp.session = session;
resp.ctime = req.ctime;
resp.ktime = now.seconds;

resp.last_req = fetch_last_request_info(client);

resp.princ_exp = 0;
resp.flags = tkt.flags;
resp.sname = service.name;
resp.realm = realm;

resp.starttime = tkt.starttime;
resp.endtime = tkt.endtime;

if (tkt.flags.RENEWABLE) then
        resp.renew_till = tkt.renew_till;
endif

pad(to cryptosystem boundary);
resp.kdc_resp_cksum(resp);
encrypt(appropriate part of response);
send(resp);
```

A.6. KRB_TGS_REP verification

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

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

decrypt(resp,session_from_tgt);
```

```
if (!integrity_ok(resp)) then
        return KRB_AP_ERR_MODIFIED;
endif
if (req.cname != resp.cname) then
        return KRB_AP_ERR_MODIFIED;
endif
if (req.realm != resp.crealm) then
        return KRB_AP_ERR_MODIFIED;
endif
if (req.sname != resp.sname) then
        return KRB_AP_ERR_MODIFIED;
endif
if (req.realm != resp.srealm) then
        return KRB_AP_ERR_MODIFIED;
endif
if (req.ctime != resp.ctime) then
        return KRB_AP_ERR_MODIFIED;
endif
if (req.addresses != resp.caddr) then
        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))
        return KRB_AP_ERR_MODIFIED;
endif

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

if ((req.kdc_options.RENEWABLE) and
    (req.rtime != 0) and (resp.renew_till > req.rtime)) then
        return KRB_AP_ERR_MODIFIED;
endif
if ((req.kdc_options.RENEWABLE-OK) and
    (resp.flags.RENEWABLE) and
    (req.till != 0) and
    (resp.renew_till > req.till)) then
        return KRB_AP_ERR_MODIFIED;
endif

if near(resp.princ_exp) then
        print(warning message);
endif
save_for_later(ticket,session,client,server,times,flags);
```

```
        check authorization_data as necessary;
```

A.7. Authenticator generation

```
        store authenticator_vno in staging area; /* authenticator_vno = 5 */
        store client name in staging area; /* cname, crealm */
        store checksum_type in staging area; /* checksum_type */
        store checksum in staging area; /* checksum */
        get system_time;
        store system_time.milliseconds in staging area; /* cmsec */
        store system_time.seconds in staging area; /* ctime */
        pad staging area to blocksize boundary; /* PAD */

        encrypt staging area;
        store encrypted data in authenticator;
        store asn1_header in authenticator; /* constant except for
                                               length encoding */
```

A.8. KRB_AP_REQ generation

```
        obtain ticket and session_key;

        store asn1_header in packet; /* constant except for length encoding */
        store protocol version in packet; /* pvno = 5 */
        store message type in packet; /* type = KRB_AP_REQ */

        if desired(MUTUAL_AUTHENTICATION) then
                set options.MUTUAL-REQUIRED;
        else
                reset options.MUTUAL-REQUIRED;
        endif
        if using_session_key then
                set options.USE-SESSION-KEY;
        else
                reset options.USE-SESSION-KEY;
        endif
        store options in packet; /* ap_options */
        store ticket in packet; /* ticket */
        generate authenticator using session_key;
        store authenticator in packet; /* authenticator */
```

A.9. KRB_AP_REQ verification

```
        receive packet;
        if packet.pvno != 5 then
                either process using other protocol spec
                or error_out(KRB_APP_ERR_BADVERSION);
        endif
        if packet.type != KRB_AP_REQ then
                error_out(KRB_APP_ERR_MSG_TYPE);
        endif
        if packet.ticket.tkt_vno != 5 then
                either process using other protocol spec
                or error_out(KRB_APP_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,etype,skvno}
else
        retrieve service key for
         packet.ticket.{sname,srealm,etype,skvno}
endif
if no_key_available then
        if cant_find_specified_skvno then
                error_out(KRB_AP_ERR_BADKEYVER);
        else
                error_out(KRB_AP_ERR_NOKEY);
        endif
endif
decrypt packet.ticket into decr_ticket using key;
if integrity_error then
        error_out(KRB_AP_BAD_INTEGRITY);
endif
decrypt packet.authenticator into decr_authenticator using
 decr_ticket.session and decr_ticket.keytype
if integrity_error then
        error_out(KRB_AP_BAD_INTEGRITY);
endif
if decr_authenticator.{cname,crealm} !=
 decr_ticket.{cname,cinst,crealm} then
        error_out(KRB_AP_ERR_BADMATCH);
endif
if sender_address(packet) is not in decr_ticket.caddr then
        error_out(KRB_AP_ERR_BADADDR);
endif
if not in_clock_skew(decr_authenticator.ctime) then
        error_out(KRB_AP_ERR_SKEW);
endif
if repeated(decr_authenticator.ctime,decr_authenticator.cmsec,
            sender_address(packet),{cname,crealm}) then
        error_out(KRB_AP_ERR_REPEAT);
endif
save_identifier(decr_authenticator.timestamp,
                decr_authenticator.cmsec,sender_address(packet),
                sender_principal(packet));
get system_time;
if decr_ticket.starttime-system_time > CLOCK_SKEW then
        /* it hasn't yet become valid */
        error_out(KRB_AP_ERR_TKT_NYV);
endif
if system_time-decr_ticket.endtime > CLOCK_SKEW then
        error_out(KRB_AP_ERR_TKT_EXPIRED);
endif
/* caller must check decr_ticket.flags for any pertinent details */
return(OK, decr_ticket, packet.ap_options.MUTUAL-REQUIRED);
```

A.10. KRB_AP_REP generation

```
store asn1_header in packet; /* constant except for length encoding */
store protocol version in packet; /* pvno = 5 */
store message type in packet; /* type = KRB_AP_REP */
store packet.ctime in staging area;
```

```
store packet.cmsec in staging area;
pad staging area to encryption blocksize boundary;
encrypt staging area using ticket.session;
store encrypted data in packet;

return packet;
```

A.11. KRB_AP_REP verification

```
receive packet;
if packet.pvno != 5 then
        either process using other protocol spec
        or error_out(KRB_APP_ERR_BADVERSION);
endif
if packet.type != KRB_AP_REQ then
        error_out(KRB_APP_ERR_MSG_TYPE);
endif
decrypted_portion = decrypt(remainder(packet));
if integrity_error then
        error_out(KRB_AP_BAD_INTEGRITY);
endif
if decrypted_portion.ctime != authenticator.system_time.ctime then
        error_out(KRB_AP_ERR_MUT_FAIL);
endif
if decrypted_portion.cmsec != authenticator.system_time.cmsec then
        error_out(KRB_AP_ERR_MUT_FAIL);
endif
return(AUTHENTICATION_SUCCEEDED);
```

A.12. KRB_SAFE generation

```
collect user data in buffer;
encode buffer as bytes_asn1;
get system time;
if sender_address > receiver_address then
        set direction bit;
else
        reset direction bit;
endif
encode host addresses as hostaddr;
/* assemble packet: */
store asn1_header in packet; /* constant except for length encoding */
store protocol version in packet; /* pvno = 5 */
store message type in packet; /* type = KRB_SAFE */
store buffer in packet; /* DATA */
store milliseconds and direction bit in packet; /* msec+D */
store host addresses in packet; /* haddr */
store timestamp in packet; /* timestamp */
store checksum type in packet; /* checksum_type */
compute checksum over packet; /* DATA to checksum_type, inclusive */
encode checksum as bytes_asn1;
store checksum in packet; /* checksum */
```

A.13. KRB_SAFE verification

```
receive packet;
if packet.pvno != 5 then
        either process using other protocol spec
        or error_out(KRB_APP_ERR_BADVERSION);
endif
if packet.type != KRB_SAFE then
        error_out(KRB_APP_ERR_MSG_TYPE);
endif
if length(packet.DATA)+length(packet.hostaddr)+
        length(packet.checksum)+10 != O/S_length(packet) then
        /* the length didn't match what the operating system
           reported */
        error_out(KRB_APP_ERR_MODIFIED);
endif
if sender_address(packet) is not in packet.hostaddr then
        /* O/S report of sender not in the list */
        error_out(KRB_APP_ERR_BADADDR);
endif
if not in_clock_skew(packet.timestamp) then
        error_out(KRB_APP_ERR_SKEW);
endif
if repeated(packet.timestamp,packet.msec,sender_address(packet),
            sender_principal(packet)) then
        error_out(KRB_APP_ERR_REPEAT);
endif
save_identifier(packet.timestamp,packet.msec,sender_address(packet),
                sender_principal(packet));
if sender_address(packet) > receiver_address(packet) then
        set computed_direction;
else
        reset computed_direction;
endif

if computed_direction != packet.direction_bit then
        error_out(KRB_APP_ERR_REPEAT); /* XXX */
endif
/* run checksum from DATA to checksum_type, inclusive */
set computed_checksum = checksum(packet);
if computed_checksum != packet.checksum then
        error_out(KRB_AP_ERR_MODIFIED);
endif
return(packet.DATA, PACKET_IS_GENUINE);
```

A.14. KRB_PRIV generation

```
collect user data in buffer;
encode buffer as bytes_asn1;
get system time;
if sender_address > receiver_address then
        set direction bit;
else
        clear direction bit;
endif
encode host addresses as hostaddr;
```

```
/* compute length of encrypted portion */
select encryption type;
add length of data buffer encoding, host address encoding, and
        6, rounding up to nearest blocksize;
/* assemble packet: */
store asn1_header in packet; /* constant except for length encoding */
store protocol version in packet; /* pvno = 5 */
store message type in packet; /* type = KRB_PRIV */
store encryption type in packet; /* etype */
store computed length of encrypted portion in packet;
store buffer in encryption area;        /* DATA */
store milliseconds and direction bit in encryption area; /* msec+D */
store host addresses in encryption area; /* haddr */
store timestamp in encryption area; /* timestamp */
encrypt data in encryption area;
store encrypted output in packet;
```

A.15. KRB_PRIV verification

```
receive packet;
if packet.pvno != 5 then
        either process using other protocol spec
        or error_out(KRB_APP_ERR_BADVERSION);
endif
if packet.type != KRB_PRIV then
        error_out(KRB_APP_ERR_MSG_TYPE);
endif
if packet.len_E + 4 != O/S_length(packet) then
        error_out(KRB_APP_ERR_MODIFIED);
endif
cleartext = decrypt(packet);
/* 14 is for pvno, type, etype, len_E, msec, timestamp */
if length(cleartext.DATA) > O/S_length(packet)-14 then
        error_out(KRB_APP_ERR_MODIFIED);
endif
/* 14 is for pvno, type, etype, len_E, msec, timestamp */
if length(cleartext.haddr) > O/S_length(packet)-14 then
        error_out(KRB_APP_ERR_MODIFIED);
endif
if length(cleartext.DATA)+length(cleartext.haddr)+
        length(packet.checksum)+14 + length(cleartext.PAD)
        != length(packet) then
        /* the length didn't match what the operating system
           reported */
        error_out(KRB_APP_ERR_MODIFIED);
endif
if sender_address(packet) is not in cleartext.haddr then
        /* O/S report of sender not in the list */
        error_out(KRB_APP_ERR_BADADDR);
endif
if not in_clock_skew(cleartext.timestamp) then
        error_out(KRB_APP_ERR_SKEW);
endif
if repeated(cleartext.timestamp,cleartext.msec,sender_address(packet),
            sender_principal(packet)) then
```

```
        error_out(KRB_APP_ERR_REPEAT);
endif
save_identifier(cleartext.timestamp,cleartext.msec,
                sender_address(packet),sender_principal(packet));
if sender_address(packet) > receiver_address(packet) then
        set computed_direction;
else
        reset computed_direction;
endif

if computed_direction != cleartext.direction_bit then
        error_out(KRB_APP_ERR_REPEAT); /* XXX */
endif
return(cleartext.DATA, PACKET_IS_GENUINE_AND_UNMODIFIED);
```

DRAFT 2

--

--

Table of Contents

Overview 1
Acknowledgements 1
1. **Introduction** 1
1.1. **Glossary of terms** 3
2. **Message Exchanges** 5
2.1. **The Authentication Service (AS) Exchange** 5
2.1.1. Generation of KRB_AS_REQ message 5
2.1.2. Receipt of KRB_AS_REQ message 6
2.1.3. Generation of KRB_AS_REP message 6
2.1.4. Generation of KRB_ERROR message 7
2.1.5. Receipt of KRB_AS_REP message 7
2.1.6. Receipt of KRB_ERROR message 7
2.2. **The Client/Server (CS) Authentication Exchange** 7
2.2.1. The KRB_AP_REQ message 7
2.2.2. Generation of a KRB_AP_REQ message 7
2.2.3. Receipt of KRB_AP_REQ message 7
2.2.4. Generation of a KRB_AP_REP message 8
2.2.5. Receipt of KRB_AP_REP message 9
2.2.6. Using the encryption key 9
2.3. **The Ticket-Granting Service (TGS) Exchange** 9
2.3.1. Generation of KRB_TGS_REQ message 9
2.3.2. Receipt of KRB_TGS_REQ message 10
2.3.3. Generation of KRB_TGS_REP message 10
2.3.4. Receipt of KRB_TGS_REP message 11
2.4. **The KRB_SAFE Exchange** 11
2.4.1. Generation of a KRB_SAFE message 11
2.4.2. Receipt of KRB_SAFE message 12
2.5. **The KRB_PRIV Exchange** 12
2.5.1. Generation of a KRB_PRIV message 12
2.5.2. Receipt of KRB_PRIV message 12
3. **Encryption** 13
3.1. **Cryptographic checksums** 13
3.2. **Checksums** 13
4. **The Kerberos Database** 13
4.1. **Database contents** 14
4.2. **Additional fields** 14
4.3. **Frequently Changing Fields** 15

4.4. **Site Constants** 15
5. **Notation** 15
5.1. **Field types** 16
5.1.1. NULL 17
5.1.2. PAD 17
5.1.3. Unsigned Integers 17
5.1.4. ASN.1 Byte vectors (bytes_asn1) 19
5.1.5. ASN.1 lengths 19
5.1.6. Strings 19
5.1.7. String Arrays 20
5.1.8. Host Addresses 20
5.2. **Predefined Data Types** 20
5.2.1. Host address types 20
5.2.2. Encryption key types 22
5.2.3. Encryption system types 23
5.2.4. Checksum types 23
6. **Field Descriptions** 24
7. **Message Specifications** 34
7.1. **Tickets and Authenticators** 34
7.1.1. Tickets 34
7.1.2. Authenticators 35
7.2. **Authentication Server (AS) message specifications** 36
7.2.1. KRB_AS_REQ definition 36
7.2.2. KRB_AS_REP definition 36
7.2.3. KRB_KDC_REP definition 36
7.3. **Client/Server (CS) message specifications** 38
7.3.1. KRB_AP_REQ definition 38
7.3.2. KRB_AP_REP definition 39
7.3.3. Error message reply 39
7.4. **Ticket-granting service (TGS) message definition** 39
7.4.1. KRB_TGS_REQ definition 39
7.4.2. KRB_TGS_REP definition 41
7.5. **KRB_SAFE message specification** 41
7.5.1. KRB_SAFE definition 41
7.6. **KRB_PRIV message specification** 42
7.6.1. KRB_PRIV definition 42
7.7. **Error message specification** 43
7.7.1. KRB_ERROR definition 43
8. **Constants** 44
9. **REFERENCES** 46
A. **Pseudo-code for protocol processing** 46
A.1. **KRB_AS_REQ generation** 46

A.2. **KRB_AS_REQ verification and KRB_AS_REP generation** 47
A.3. **KRB_AS_REP verification** 50
A.4. **KRB_TGS_REQ generation** 51
A.5. **KRB_TGS_REQ verification and KRB_TGS_REP generation** 52
A.6. **KRB_TGS_REP verification** 56
A.7. **Authenticator generation** 58
A.8. **KRB_AP_REQ generation** 58
A.9. **KRB_AP_REQ verification** 58
A.10. **KRB_AP_REP generation** 59
A.11. **KRB_AP_REP verification** 60
A.12. **KRB_SAFE generation** 60
A.13. **KRB_SAFE verification** 61
A.14. **KRB_PRIV generation** 61
A.15. **KRB_PRIV verification** 62