NAME

kerberos, krb_mk_req, krb_rd_req, krb_kntoln, krb_set_key, krb_get_cred, krb_mk_safe, krb_rd_safe, krb_mk_err, krb_rd_err – Kerberos authentication library

SYNOPSIS

cc [*flag* ...] *file* ... –**lkrb** [*library* ...]

#include <kerberos/krb.h>

extern char ∗**krb_err_txt[ ];**

int krb_mk_req(KTEXT *authent*, **const char** ∗*service*, **const char** ∗*instance*,
    const char ∗*realm*, **const u_long** *checksum*);

int krb_rd_req(const KTEXT *authent*, **const char** ∗ *service*, **char** ∗ *instance*,
    const u_long *from_addr*, **AUTH_DAT** ∗*ad*, **const char** ∗*fn*);

int krb_kntoln(const AUTH_DAT ∗*ad*, **char** ∗*lname*);

int krb_set_key(const char ∗*key*, **const int** *cvt*);

int krb_get_cred(const char ∗*service*, **const char** ∗*instance*, **const char** ∗*realm*,
    CREDENTIALS ∗*c*);

long krb_mk_safe(const u_char ∗*in*, **u_char** ∗*out*, **const u_long** *in_length*,
    const des_cblock *key*, **const struct sockaddr_in** *sender*,
    const struct sockaddr_in *receiver*);

long krb_rd_safe(const u_char ∗*in*, **const u_long** *length*, **const des_cblock** *key*,
    const struct sockaddr_in *sender*, **const struct sockaddr_in** *receiver*,
    MSG_DAT ∗*msg_data*);

long krb_mk_err(u_char ∗*out*, **const long** *code*, **const char** ∗*string*);

long krb_rd_err(const u_char ∗*in*, **const u_long** *length*, **long** ∗*code*,
    MSG_DAT *msg_data*);

MT-LEVEL

Unsafe

DESCRIPTION

This library supports network authentication and various related operations. The library contains many routines beyond those described in this man page, but they are not intended to be used directly. Instead, they are called by the routines that are described, the authentication server and the login program.

krb_err_txt[ ] contains text string descriptions of various Kerberos error codes returned by some of the routines below.

krb_mk_req( ) takes a pointer to a text structure in which an authenticator is to be built. It also takes the name, instance, and realm of the service to be used and an optional checksum. It is up to the application to decide how to generate the checksum. **krb_mk_req( )** then retrieves a ticket for the desired service and creates an authenticator. The authenticator is built in *authent* and is accessible to the calling procedure.

It is up to the application to get the authenticator to the service where it will be read by **krb_rd_req( )**. Unless an attacker possesses the session key contained in the ticket, it will be unable to modify the authenticator. Thus, the checksum can be used to verify the authenticity of the other data that will pass through a connection.

krb_mk_req( ) returns KSUCCESS if successful, otherwise a Kerberos error code as defined in **<kerberos/krb.h>**.

krb_rd_req( ) takes an authenticator of type KTEXT, a service name, an instance, the address of the host originating the request, and a pointer to a structure of type AUTH_DAT which is filled in with information obtained from the authenticator. It also optionally takes the name of the file in which it will find the secret key(s) for the service. If the supplied *instance* is "*", then the first service key with the same service name found in the service key file will be used, and the *instance* argument will be filled in with the chosen instance. This means that the caller must provide space for such an instance name.

If the last argument is the null string (" "), **krb_rd_req( )** will use the file **/etc/srvtab** to find its keys. If the last argument is NULL, it will assume that the key has been set by **krb_set_key( )** and will not bother looking further.

krb_rd_req( ) is used to find out information about the principal when a request has been made to a service. It is up to the application protocol to get the authenticator from the client to the service. The authenticator is then passed to **krb_rd_req( )** to extract the desired information.

krb_rd_req( ) returns zero (RD_AP_OK) upon successful authentication. If a packet was forged, modified, or replayed, authentication will fail. If the authentication fails, a non-zero value is returned indicating the particular problem encountered. See **<kerberos/krb.h>** for the list of error codes.

krb_kntoln( ) converts a Kerberos name to a local name. It takes a structure of type AUTH_DAT and uses the name, instance, and realm to determine the corresponding local name. A valid local name is returned if the instance is NULL and the realm is the same as the local realm. The local name returned is the Kerberos name and can be used by an application to change uids, directories, or other parameters. This routine is not an integral part of Kerberos, but is provided to support the use of Kerberos in existing utilities. This routine returns KSUCCESS or KFAILURE.

krb_set_key( ) takes as an argument a DES key. It then creates a key schedule from it and saves the original key to be used as an initialization vector. It is used to set the server's key which must be used to decrypt tickets.

If called with a non-zero second argument, **krb_set_key( )** will first convert the input from a string of arbitrary length to a DES key by encrypting it with a one-way function.

In most cases it should not be necessary to call **krb_set_key( )**. The necessary keys will usually be obtained and set inside **krb_rd_req( )**. **krb_set_key( )** is provided for those applications that do not wish to place the application keys on disk. It returns 0 for success, otherwise a non-zero value.

krb_get_cred( ) searches the caller's ticket file for a ticket for the given *service*, *instance*, and *realm*. If a ticket is found, the given CREDENTIALS structure is filled in with the ticket information.

If the ticket was found, **krb_get_cred( )** returns GC_OK. If the ticket file cannot be found, cannot be read, does not belong to the user (other than root), is not a regular file, or is in the wrong mode, the error GC_TKFIL is returned.

krb_mk_safe( ) creates an authenticated, but unencrypted message from any arbitrary application data, pointed to by *in* and *in_length* bytes long. The private session key, pointed to by *key,* is used to seed the **quad_cksum( )** checksum algorithm used as part of the authentication. *sender* and *receiver* point to the Internet address of the two parties. This message does not provide privacy, but does protect (via detection) against modifications, insertions or replays. The encapsulated message and header are placed in the area pointed to by *out* and the routine returns the length of the output, or −1 indicating an error.

krb_rd_safe( ) authenticates a received **krb_mk_safe( )** message. *in* points to the beginning of the received message, whose length is specified in *in_length*. The private session key, pointed to by *key,* is used to seed the **quad_cksum( )** routine as part of the authentication. *msg_data* is a pointer to a MSG_DAT struct, defined in **<kerberos/krb.h>**. The routine fills in these MSG_DAT fields: the *app_data* field with a pointer to the application data, *app_length* with the length of the *app_data* field, *time_sec* and *time_5ms* with the timestamps in the message, and *swap* with a 1 if the byte order of the receiver is different than that of the sender. (The application must still determine if it is appropriate to byte-swap application data; the Kerberos protocol fields are already taken care of.)

The routine returns zero if successful, or a Kerberos error code. Modified messages and old messages cause errors, but it is up to the caller to check the time sequence of messages, and to check against recently replayed messages.

krb_mk_err( ) constructs an application level error message that may be used along with **krb_mk_safe( )**. *out* is a pointer to the output buffer, *code* is an application specific error code, and *string* is an application specific error string. This routine returns the length of the error reply.

krb_rd_err( ) unpacks a received **krb_mk_err( )** message. *in* points to the beginning of the received message, whose length is specified in *in_length*. *code* is a pointer to a value to be filled in with the error value provided by the application. *msg_data* is a pointer to a MSG_DAT struct, defined in **<kerberos/krb.h>**. The routine fills in these MSG_DAT fields: the *app_data* field with a pointer to the application error text, *app_length* with the length of the *app_data* field, and *swap* with a 1 if the byte order of the receiver is different than that of the sender. (The application must still determine if it is appropriate to byte-swap application data; the Kerberos protocol fields are already taken care of).

The routine returns zero if the error message has been successfully received, or a Kerberos error code.

The KTEXT structure is used to pass around text of varying lengths. It consists of a buffer for the data, and a length. **krb_rd_req( )** takes an argument of this type containing the authenticator, and **krb_mk_req( )** returns the authenticator in a structure of this type. KTEXT itself is really a pointer to the structure. The actual structure is of type KTEXT_ST.

The AUTH_DAT structure is filled in by **krb_rd_req( )**. It must be allocated before calling **krb_rd_req( )**, and a pointer to it is passed. The structure is filled in with data obtained from Kerberos. The MSG_DAT structure is filled in by either **krb_rd_safe( )** or **krb_rd_err( )**. It must be allocated before the call and a pointer to it is passed. The structure is filled in with data obtained from Kerberos.

FILES

/usr/lib/libkrb.*
/etc/aname
/etc/srvtab
/tmp/tkt*uid*

SEE ALSO

kerberos(1), **kerberos_rpc**(3N), **krb_realmofhost**(3N), **krb_sendauth**(3N), **krb_set_tkt_string**(3N), **krb.conf**(4), **krb.realms**(4)

NOTES

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

BUGS

The caller of **krb_rd_req( )** and **krb_rd_safe( )** must check time order and for replay attempts.

AUTHORS

Clifford Neuman, MIT Project Athena
Steve Miller, MIT Project Athena / Digital Equipment Corporation

RESTRICTIONS