Kerberos V5 application programming library

MIT Information Systems

September 10, 1996

Contents

1 Introduction

This document describes the routines that make up the Kerberos V5 application programming interface. It is geared towards programmers who already have a basic familiarity with Kerberos and are in the process of including Kerberos authentication as part of applications being developed.

The function descriptions included are up to date, even if the description of the functions may be hard to understand for the novice Kerberos programmer.

1.1 Acknowledgments

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

Version 5 (described in this document) has evolved from Version 4 based on new requirements and desires for features not available in Version 4.

1.2 Kerberos Basics

Kerberos performs authentication as a trusted third-party authentication service by using conventional (shared secret key[1]) cryptography. Kerberos provides a means of verifying the identities of principals, without relying on authentication by the host operating system, without basing trust on host addresses, without requiring physical

[1] *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.

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.

When integrating Kerberos into an application it is important to review how and when Kerberos functions are used to ensure that the application's design does not compromise the authentication. For instance, an application which uses Kerberos' functions only upon the *initiation* of a stream-based network connection, and assumes the absence of any active attackers who might be able to "hijack" the stream connection.

The Kerberos protocol code libraries, whose API is described in this document, can be used to provide encryption to any application. In order to add authentication to its transactions, a typical network application adds one or two calls to the Kerberos library, which results in the transmission of the necessary messages to achieve authentication.

The two methods for obtaining credentials, the initial ticket exchange and the ticket granting ticket exchange, use slightly different protocols and require different API routines. The basic difference an API programmer will see is that the initial request does not require a ticket granting ticket (TGT) but does require the client's secret key because the reply is sent back encrypted in the client's secret key. Usually this request is for a TGT and TGT based exchanges are used from then on. In a TGT exchange the TGT is sent as part of the request for tickets and the reply is encrypted in the session key from the TGT. For example, once a user's password is used to obtain a TGT, it is not required for subsequent TGT exchanges.

The reply consists of a ticket and a session key, encrypted either in the user's secret key (i.e., password), or the TGT session key. The combination of a ticket and a session key is known as a set of *credentials*.[2] An application client can use these credentials to authenticate to the application server by sending the ticket and an *authenticator* to the server. The authenticator is encrypted in the session key of the ticket, and contains the name of the client, the name of the server, the time the authenticator was created.

In order to verify the authentication, the application server decrypts the ticket using its service key, which is only known by the application server and the Kerberos server. Inside the ticket, the Kerberos server had placed the name of the client, the name of the server, a DES key associated with this ticket, and some additional information. The application server then uses the ticket session key to decrypt the authenticator, and verifies that the information in the authenticator matches the information in the ticket, and that the timestamp in the authenticator is recent (to prevent reply attacks). Since the session key was generated randomly by the Kerberos server, and delivered only encrypted in the service key, and in a key known only by the user, the application server can be confident that user is really who he or she claims to be, by virtue of the fact that the user was able to encrypt the authenticator in the correct key.

To provide detection of both replay attacks and message stream modification attacks, the integrity of all the messages exchanged between principals can also be guaranteed[3] by generating and transmitting a collision-proof checksum[4] of the client's message, keyed with the session key. Privacy and integrity of the messages exchanged between principals can be secured[5] by encrypting the data to be passed using the session key.

[2]In Kerberos V4, the "ticket file" was a bit of a misnomer, since it contained both tickets and their associated session keys. In Kerberos V5, the "ticket file" has been renamed to be the *credentials cache*.

[3]Using **krb5_mk_safe**() and **krb5_rd_safe**() to create and verify KRB5_SAFE messages

[4]aka cryptographic checksum, elsewhere this is called a hash or digest function

[5]Using **krb5_mk_priv**() and **krb5_rd_priv**() to create and verify KRB5_PRIV messages

1.2.1 The purpose of Realms

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

By establishing *inter-realm* keys, the administrators of two realms can allow a client authenticated in the local realm to use its credentials remotely. The exchange of inter-realm keys (a separate key may be used for each direction) registers the ticket-granting service of each realm as a principal in the other realm. A client is then able to obtain a ticket-granting ticket for the remote realm's ticket-granting service from its local realm. When that ticket-granting ticket is used, the remote ticket-granting service uses the inter-realm key (which usually differs from its own normal TGS key) to decrypt the ticket-granting ticket, and is thus certain that it was issued by the client's own TGS. Tickets issued by the remote ticket-granting service will indicate to the end-service that the client was authenticated from another realm.

This method can be repeated to authenticate throughout an organization across multiple realms. To build a valid authentication path[6] to a distant realm, the local realm must share an inter-realm key with an intermediate realm which communicates[7] with either the distant remote realm or yet another intermediate realm.

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

Although realms are typically hierarchical, intermediate realms may be bypassed to achieve cross-realm authentication through alternate authentication paths[8]. It is important for the end-service to know which realms were transited when deciding how much faith to place in the authentication process. To facilitate this decision, a field in each ticket contains the names of the realms that were involved in authenticating the client.

1.2.2 Fundamental assumptions about the environment

Kerberos has certain limitations that should be kept in mind when designing security measures:

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

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

[6] An *authentication path* is the sequence of intermediate realms that are transited in communicating from one realm to another.

[7] A realm is said to *communicate* with another realm if the two realms share an inter-realm key

[8] These might be established to make communication between two realms more efficient

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

1.3 Glossary of terms

Below is a list of terms used throughout this document.

Authentication Verifying the claimed identity of a principal.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2 Useful KDC parameters to know about

The following is a list of options which can be passed to the Kerberos server (also known as the Key Distribution Center or KDC). These options affect what sort of tickets the KDC will return to the application program. The KDC options can be passed to **krb5_get_in_tkt**(), **krb5_get_in_tkt_with_password**(), **krb5_get_in_tkt_with_skey**(), and **krb5_send_tgs**().

Symbol	RFC section	Valid for get_in_tkt?
KDC_OPT_FORWARDABLE	2.6	yes
KDC_OPT_FORWARDED	2.6	
KDC_OPT_PROXIABLE	2.5	yes
KDC_OPT_PROXY	2.5	
KDC_OPT_ALLOW_POSTDATE	2.4	yes
KDC_OPT_POSTDATED	2.4	yes
KDC_OPT_RENEWABLE	2.3	yes
KDC_OPT_RENEWABLE_OK	2.7	yes
KDC_OPT_ENC_TKT_IN_SKEY	2.7	
KDC_OPT_RENEW	2.3	
KDC_OPT_VALIDATE	2.2	

The following is a list of preauthentication methods which are supported by Kerberos. Most preauthentication methods are used by **krb5_get_in_tkt**(), **krb5_get_in_tkt_with_password**(), and **krb5_get_in_tkt_with_skey**(); at some sites, the Kerberos server can be configured so that during the initial ticket transation, it will only return encrypted tickets after the user has proven his or her identity using a supported preauthentication mechanism. This is done to make certain password guessing attacks more difficult to carry out.

Symbol	In RFC?	Valid for get_in_tkt?
KRB5_PADATA_NONE	yes	yes
KRB5_PADATA_AP_REQ	yes	
KRB5_PADATA_TGS_REQ	yes	
KRB5_PADATA_PW_SALT	yes	
KRB5_PADATA_ENC_TIMESTAMP	yes	yes
KRB5_PADATA_ENC_SECURID		yes

KRB5_PADATA_TGS_REQ is rarely used by a programmer; it is used to pass the ticket granting ticket to the Ticket Granting Service (TGS) during a TGS transaction (as opposed to an initial ticket transaction).

KRB5_PW_SALT is not really a preauthentication method at all. It is passed back from the Kerberos server to application program, and it contains a hint to the proper password salting algorithm which should be used during the initial ticket exchange.

3 Error tables

3.1 error_table krb5

The Kerberos v5 library error code table follows. Protocol error codes are ERROR_TABLE_BASE_krb5 + the protocol error code number. Other error codes start at ERROR_TABLE_BASE_krb5 + 128.

KRB5KDC_ERR_NONE	No error
KRB5KDC_ERR_NAME_EXP	Client's entry in database has expired
KRB5KDC_ERR_SERVICE_EXP	Server's entry in database has expired
KRB5KDC_ERR_BAD_PVNO	Requested protocol version not supported
KRB5KDC_ERR_C_OLD_MAST_KVNO	Client's key is encrypted in an old master key
KRB5KDC_ERR_S_OLD_MAST_KVNO	Server's key is encrypted in an old master key
KRB5KDC_ERR_C_PRINCIPAL_UNKNOWN	Client not found in Kerberos database
KRB5KDC_ERR_S_PRINCIPAL_UNKNOWN	Server not found in Kerberos database
KRB5KDC_ERR_PRINCIPAL_NOT_UNIQUE	Principal has multiple entries in Kerberos database
KRB5KDC_ERR_NULL_KEY	Client or server has a null key
KRB5KDC_ERR_CANNOT_POSTDATE	Ticket is ineligible for postdating
KRB5KDC_ERR_NEVER_VALID	Requested effective lifetime is negative or too short
KRB5KDC_ERR_POLICY	KDC policy rejects request
KRB5KDC_ERR_BADOPTION	KDC can't fulfill requested option
KRB5KDC_ERR_ETYPE_NOSUPP	KDC has no support for encryption type
KRB5KDC_ERR_SUMTYPE_NOSUPP	KDC has no support for checksum type
KRB5KDC_ERR_PADATA_TYPE_NOSUPP	KDC has no support for padata type
KRB5KDC_ERR_TRTYPE_NOSUPP	KDC has no support for transited type
KRB5KDC_ERR_CLIENT_REVOKED	Clients credentials have been revoked
KRB5KDC_ERR_SERVICE_REVOKED	Credentials for server have been revoked
KRB5KDC_ERR_TGT_REVOKED	TGT has been revoked
KRB5KDC_ERR_CLIENT_NOTYET	Client not yet valid - try again later
KRB5KDC_ERR_SERVICE_NOTYET	Server not yet valid - try again later
KRB5KDC_ERR_KEY_EXP	Password has expired
KRB5KDC_PREAUTH_FAILED	Preauthentication failed
KRB5KDC_ERR_PREAUTH_REQUIRE	Additional pre-authentication required
KRB5KDC_ERR_SERVER_NOMATCH	Requested server and ticket don't match
error codes 27-30 are currently placeholders	

KRB5KRB_AP_ERR_BAD_INTEGRITY	Decrypt integrity check failed
KRB5KRB_AP_ERR_TKT_EXPIRED	Ticket expired
KRB5KRB_AP_ERR_TKT_NYV	Ticket not yet valid
KRB5KRB_AP_ERR_REPEAT	Request is a replay
KRB5KRB_AP_ERR_NOT_US	The ticket isn't for us
KRB5KRB_AP_ERR_BADMATCH	Ticket/authenticator don't match
KRB5KRB_AP_ERR_SKEW	Clock skew too great
KRB5KRB_AP_ERR_BADADDR	Incorrect net address
KRB5KRB_AP_ERR_BADVERSION	Protocol version mismatch
KRB5KRB_AP_ERR_MSG_TYPE	Invalid message type
KRB5KRB_AP_ERR_MODIFIED	Message stream modified
KRB5KRB_AP_ERR_BADORDER	Message out of order
KRB5PLACEHOLD_43	KRB5 error code 43
KRB5KRB_AP_ERR_BADKEYVER	Key version is not available
KRB5KRB_AP_ERR_NOKEY	Service key not available
KRB5KRB_AP_ERR_MUT_FAIL	Mutual authentication failed
KRB5KRB_AP_ERR_BADDIRECTION	Incorrect message direction
KRB5KRB_AP_ERR_METHOD	Alternative authentication method required
KRB5KRB_AP_ERR_BADSEQ	Incorrect sequence number in message
KRB5KRB_AP_ERR_INAPP_CKSUM	Inappropriate type of checksum in message
error codes 51-59 are currently placeholders	
KRB5KRB_ERR_GENERIC	Generic error (see e-text)
KRB5KRB_ERR_FIELD_TOOLONG	Field is too long for this implementation
error codes 62-127 are currently placeholders	

KRB5_LIBOS_BADLOCKFLAG	Invalid flag for file lock mode
KRB5_LIBOS_CANTREADPWD	Cannot read password
KRB5_LIBOS_BADPWDMATCH	Password mismatch
KRB5_LIBOS_PWDINTR	Password read interrupted
KRB5_PARSE_ILLCHAR	Illegal character in component name
KRB5_PARSE_MALFORMED	Malformed representation of principal
KRB5_CONFIG_CANTOPEN	Can't open/find configuration file
KRB5_CONFIG_BADFORMAT	Improper format of configuration file
KRB5_CONFIG_NOTENUFSPACE	Insufficient space to return complete information
KRB5_BADMSGTYPE	Invalid message type specified for encoding
KRB5_CC_BADNAME	Credential cache name malformed
KRB5_CC_UNKNOWN_TYPE	Unknown credential cache type
KRB5_CC_NOTFOUND	Matching credential not found
KRB5_CC_END	End of credential cache reached
KRB5_NO_TKT_SUPPLIED	Request did not supply a ticket
KRB5KRB_AP_WRONG_PRINC	Wrong principal in request
KRB5KRB_AP_ERR_TKT_INVALID	Ticket has invalid flag set
KRB5_PRINC_NOMATCH	Requested principal and ticket don't match
KRB5_KDCREP_MODIFIED	KDC reply did not match expectations
KRB5_KDCREP_SKEW	Clock skew too great in KDC reply
KRB5_IN_TKT_REALM_MISMATCH	Client/server realm mismatch in initial ticket requst
KRB5_PROG_ETYPE_NOSUPP	Program lacks support for encryption type
KRB5_PROG_KEYTYPE_NOSUPP	Program lacks support for key type
KRB5_WRONG_ETYPE	Requested encryption type not used in message
KRB5_PROG_SUMTYPE_NOSUPP	Program lacks support for checksum type
KRB5_REALM_UNKNOWN	Cannot find KDC for requested realm
KRB5_SERVICE_UNKNOWN	Kerberos service unknown
KRB5_KDC_UNREACH	Cannot contact any KDC for requested realm
KRB5_NO_LOCALNAME	No local name found for principal name

KRB5_RC_TYPE_EXISTS	Replay cache type is already registered
KRB5_RC_MALLOC	No more memory to allocate (in replay cache code)
KRB5_RC_TYPE_NOTFOUND	Replay cache type is unknown
KRB5_RC_UNKNOWN	Generic unknown RC error
KRB5_RC_REPLAY	Message is a replay
KRB5_RC_IO	Replay I/O operation failed XXX
KRB5_RC_NOIO	Replay cache type does not support non-volatile storage
KRB5_RC_PARSE	Replay cache name parse/format error
KRB5_RC_IO_EOF	End-of-file on replay cache I/O
KRB5_RC_IO_MALLOC	No more memory to allocate (in replay cache I/O code)
KRB5_RC_IO_PERM	Permission denied in replay cache code
KRB5_RC_IO_IO	I/O error in replay cache i/o code
KRB5_RC_IO_UNKNOWN	Generic unknown RC/IO error
KRB5_RC_IO_SPACE	Insufficient system space to store replay information
KRB5_TRANS_CANTOPEN	Can't open/find realm translation file
KRB5_TRANS_BADFORMAT	Improper format of realm translation file
KRB5_LNAME_CANTOPEN	Can't open/find lname translation database
KRB5_LNAME_NOTRANS	No translation available for requested principal
KRB5_LNAME_BADFORMAT	Improper format of translation database entry
KRB5_CRYPTO_INTERNAL	Cryptosystem internal error
KRB5_KT_BADNAME	Key table name malformed
KRB5_KT_UNKNOWN_TYPE	Unknown Key table type
KRB5_KT_NOTFOUND	Key table entry not found
KRB5_KT_END	End of key table reached
KRB5_KT_NOWRITE	Cannot write to specified key table
KRB5_KT_IOERR	Error writing to key table
KRB5_NO_TKT_IN_RLM	Cannot find ticket for requested realm
KRB5DES_BAD_KEYPAR	DES key has bad parity
KRB5DES_WEAK_KEY	DES key is a weak key
KRB5_BAD_KEYTYPE	Keytype is incompatible with encryption type
KRB5_BAD_KEYSIZE	Key size is incompatible with encryption type
KRB5_BAD_MSIZE	Message size is incompatible with encryption type
KRB5_CC_TYPE_EXISTS	Credentials cache type is already registered.
KRB5_KT_TYPE_EXISTS	Key table type is already registered.
KRB5_CC_IO	Credentials cache I/O operation failed XXX
KRB5_FCC_PERM	Credentials cache file permissions incorrect
KRB5_FCC_NOFILE	No credentials cache file found
KRB5_FCC_INTERNAL	Internal file credentials cache error
KRB5_CC_NOMEM	No more memory to allocate (in credentials cache code)

errors for dual TGT library calls

KRB5_INVALID_FLAGS	Invalid KDC option combination (library internal error)
KRB5_NO_2ND_TKT	Request missing second ticket
KRB5_NOCREDS_SUPPLIED	No credentials supplied to library routine

errors for sendauth and recvauth

KRB5_SENDAUTH_BADAUTHVERS	Bad sendauth version was sent
KRB5_SENDAUTH_BADAPPLVERS	Bad application version was sent (via sendauth)
KRB5_SENDAUTH_BADRESPONSE	Bad response (during sendauth exchange)
KRB5_SENDAUTH_REJECTED	Server rejected authentication (during sendauth exchange)
KRB5_SENDAUTH_MUTUAL_FAILED	Mutual authentication failed (during sendauth exchange)

errors for preauthentication	
KRB5_PREAUTH_BAD_TYPE	Unsupported preauthentication type
KRB5_PREAUTH_NO_KEY	Required preauthentication key not supplied
KRB5_PREAUTH_FAILED	Generic preauthentication failure

version number errors	
KRB5_RCACHE_BADVNO	Unsupported replay cache format version number
KRB5_CCACHE_BADVNO	Unsupported credentials cache format version number
KRB5_KEYTAB_BADVNO	Unsupported key table format version number

other errors	
KRB5_PROG_ATYPE_NOSUPP	Program lacks support for address type
KRB5_RC_REQUIRED	Message replay detection requires rcache parameter
KRB5_ERR_BAD_HOSTNAME	Hostname cannot be canonicalized
KRB5_ERR_HOST_REALM_UNKNOWN	Cannot determine realm for host
KRB5_SNAME_UNSUPP_NAMETYPE	Conversion to service principal undefined for name type
KRB5KRB_AP_ERR_V4_REPLY	Initial Ticket Response appears to be Version 4 error
KRB5_REALM_CANT_RESOLVE	Cannot resolve KDC for requested realm
KRB5_TKT_NOT_FORWARDABLE	Requesting ticket can't get forwardable tickets

3.2 error_table kdb5

The Kerberos v5 database library error code table

From the server side routines	
KRB5_KDB_INUSE	Entry already exists in database
KRB5_KDB_UK_SERROR	Database store error
KRB5_KDB_UK_RERROR	Database read error
KRB5_KDB_UNAUTH	Insufficient access to perform requested operation
KRB5_KDB_NOENTRY	No such entry in the database
KRB5_KDB_ILL_WILDCARD	Illegal use of wildcard
KRB5_KDB_DB_INUSE	Database is locked or in use–try again later
KRB5_KDB_DB_CHANGED	Database was modified during read
KRB5_KDB_TRUNCATED_RECORD	Database record is incomplete or corrupted
KRB5_KDB_RECURSIVELOCK	Attempt to lock database twice
KRB5_KDB_NOTLOCKED	Attempt to unlock database when not locked
KRB5_KDB_BADLOCKMODE	Invalid kdb lock mode
KRB5_KDB_DBNOTINITED	Database has not been initialized
KRB5_KDB_DBINITED	Database has already been initialized
KRB5_KDB_ILLDIRECTION	Bad direction for converting keys
KRB5_KDB_NOMASTERKEY	Cannot find master key record in database
KRB5_KDB_BADMASTERKEY	Master key does not match database
KRB5_KDB_INVALIDKEYSIZE	Key size in database is invalid
KRB5_KDB_CANTREAD_STORED	Cannot find/read stored master key
KRB5_KDB_BADSTORED_MKEY	Stored master key is corrupted
KRB5_KDB_CANTLOCK_DB	Insufficient access to lock database
KRB5_KDB_DB_CORRUPT	Database format error
KRB5_KDB_BAD_VERSION	Unsupported version in database entry

3.3 error_table kv5m

The Kerberos v5 magic numbers errorcode table follows. These are used for the magic numbers found in data structures.

KV5M_NONE	Kerberos V5 magic number table
KV5M_PRINCIPAL	Bad magic number for krb5_principal structure
KV5M_DATA	Bad magic number for krb5_data structure
KV5M_KEYBLOCK	Bad magic number for krb5_keyblock structure
KV5M_CHECKSUM	Bad magic number for krb5_checksum structure
KV5M_ENCRYPT_BLOCK	Bad magic number for krb5_encrypt_block structure
KV5M_ENC_DATA	Bad magic number for krb5_enc_data structure
KV5M_CRYPTOSYSTEM_ENTRY	Bad magic number for krb5_cryptosystem_entry structure
KV5M_CS_TABLE_ENTRY	Bad magic number for krb5_cs_table_entry structure
KV5M_CHECKSUM_ENTRY	Bad magic number for krb5_checksum_entry structure
KV5M_AUTHDATA	Bad magic number for krb5_authdata structure
KV5M_TRANSITED	Bad magic number for krb5_transited structure
KV5M_ENC_TKT_PART	Bad magic number for krb5_enc_tkt_part structure
KV5M_TICKET	Bad magic number for krb5_ticket structure
KV5M_AUTHENTICATOR	Bad magic number for krb5_authenticator structure
KV5M_TKT_AUTHENT	Bad magic number for krb5_tkt_authent structure
KV5M_CREDS	Bad magic number for krb5_creds structure
KV5M_LAST_REQ_ENTRY	Bad magic number for krb5_last_req_entry structure
KV5M_PA_DATA	Bad magic number for krb5_pa_data structure
KV5M_KDC_REQ	Bad magic number for krb5_kdc_req structure
KV5M_ENC_KDC_REP_PART	Bad magic number for krb5_enc_kdc_rep_part structure
KV5M_KDC_REP	Bad magic number for krb5_kdc_rep structure
KV5M_ERROR	Bad magic number for krb5_error structure
KV5M_AP_REQ	Bad magic number for krb5_ap_req structure
KV5M_AP_REP	Bad magic number for krb5_ap_rep structure
KV5M_AP_REP_ENC_PART	Bad magic number for krb5_ap_rep_enc_part structure
KV5M_RESPONSE	Bad magic number for krb5_response structure
KV5M_SAFE	Bad magic number for krb5_safe structure
KV5M_PRIV	Bad magic number for krb5_priv structure
KV5M_PRIV_ENC_PART	Bad magic number for krb5_priv_enc_part structure
KV5M_CRED	Bad magic number for krb5_cred structure
KV5M_CRED_INFO	Bad magic number for krb5_cred_info structure
KV5M_CRED_ENC_PART	Bad magic number for krb5_cred_enc_part structure
KV5M_PWD_DATA	Bad magic number for krb5_pwd_data structure
KV5M_ADDRESS	Bad magic number for krb5_address structure
KV5M_KEYTAB_ENTRY	Bad magic number for krb5_keytab_entry structure
KV5M_CONTEXT	Bad magic number for krb5_context structure
KV5M_OS_CONTEXT	Bad magic number for krb5_os_context structure

3.4 error_table asn1

The Kerberos v5/ASN.1 error table mappings

ASN1_BAD_TIMEFORMAT	ASN.1 failed call to system time library
ASN1_MISSING_FIELD	ASN.1 structure is missing a required field
ASN1_MISPLACED_FIELD	ASN.1 unexpected field number
ASN1_TYPE_MISMATCH	ASN.1 type numbers are inconsistent
ASN1_OVERFLOW	ASN.1 value too large
ASN1_OVERRUN	ASN.1 encoding ended unexpectedly
ASN1_BAD_ID	ASN.1 identifier doesn't match expected value
ASN1_BAD_LENGTH	ASN.1 length doesn't match expected value
ASN1_BAD_FORMAT	ASN.1 badly-formatted encoding
ASN1_PARSE_ERROR	ASN.1 parse error

4 libkrb5.a functions

This section describes the functions provided in the `libkrb5.a` library. The library is built from several pieces, mostly for convenience in programming, maintenance, and porting.

4.1 Main functions

The main functions deal with the nitty-gritty details: verifying tickets, creating authenticators, and the like.

4.1.1 The krb5_context

The **krb5_context** is designed to represent the per process state. When the library is made thread-safe, the context will represent the per-thread state. Global parameters which are "context" specific are stored in this structure, including default realm, default encryption type, default configuration files and the like. Functions exist to provide full access to the data structures stored in the context and should not be accessed directly by developers.

init_context

krb5_error_code
krb5_init_context(*/* IN/OUT */*
krb5_context * `context`)

Initializes the context `*context` for the application. Currently the context contains the encryption types, a pointer to operating specific data and the default realm. In the future, the context may be also contain thread specific data. The data in the context should be freed with **krb5_free_context**().

Returns system errors.

free_context

void
krb5_free_context(*/* IN/OUT */*
krb5_context `context`)

Frees the context returned by **krb5_init_context**(). Internally calls **krb5_os_free_context**().

set_default_in_tkt_etypes

krb5_error_code
krb5_set_default_in_tkt_etypes(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_enctype * `etypes`)

Sets the desired default encryption type `etypes` for the context if valid.

Returns ENOMEM, KRB5_PROG_ETYPE_NOSUPP.

krb5_error_code get_default_in_tkt_etypes
krb5_get_default_in_tkt_etypes(*/* IN/OUT */*
krb5_context `context`,
/ OUT */*
krb5_enctype ** `etypes`)

Retrieves the default encryption types from the context and stores them in etypes which should be freed by the caller.

Returns ENOMEM.

4.1.2 The krb5_auth_context

While the **krb5_context** represents a per-process or per-thread context, the **krb5_auth_context** represents a per-connection context are are used by the various functions involved directly in client/server authentication. Some of the data stored in this context include keyblocks, addresses, sequence numbers, authenticators, checksum type, and replay cache pointer.

krb5_error_code auth_con_init
krb5_auth_con_init(*/* IN/OUT */*
krb5_context `context`,
/ OUT */*
krb5_auth_context * `auth_context`)

The auth_context may be described as a per connection context. This context contains all data pertinent to the the various authentication routines. This function initializes the auth_context.

The default flags for the context are set to enable the use of the replay cache (KRB5_AUTH_CONTEXT_DO_TIME) but no sequence numbers. The function **krb5_auth_con_setflags**() allows the flags to be changed.

The default checksum type is set to CKSUMTYPE_RSA_MD4_DES. This may be changed with **krb5_auth_con_setcksumtype**().

The auth_context structure should be freed with **krb5_auth_con_free**().

krb5_error_code auth_con_free
krb5_auth_con_free(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`)

Frees the auth_context `auth_context` returned by **krb5_auth_con_init**().

krb5_error_code auth_con_setflags
krb5_auth_con_setflags(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_int32 `flags`)

Sets the flags of `auth_context` to funcparamflags. Valid flags are:

Symbol	Meaning
KRB5_AUTH_CONTEXT_DO_TIME	Use timestamps
KRB5_AUTH_CONTEXT_RET_TIME	Save timestamps to output structure
KRB5_AUTH_CONTEXT_DO_SEQUENCE	Use sequence numbers
KRB5_AUTH_CONTEXT_RET_SEQUENCE	Copy sequence numbers to output structure

auth_con_getflags

krb5_error_code
krb5_auth_con_getflags(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_auth_context `auth_context`,
/ OUT */*
krb5_int32 * `flags`)

Retrievs the flags of `auth_context`.

auth_con_setaddrs

krb5_error_code
krb5_auth_con_setaddrs(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_address * `local_addr`,
krb5_address * `remote_addr`)

Copies the `local_addr` and `remote_addr` into the `auth_context`. If either address is NULL, the previous address remains in place.

auth_con_getaddrs

krb5_error_code
krb5_auth_con_getaddrs(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ OUT */*
krb5_address ** `local_addr`,
krb5_address ** `remote_addr`)

Retrieves `local_addr` and `remote_addr` from the `auth_context`. If `local_addr` or `remote_addr` is not NULL, the memory is first freed with **krb5_free_address**() and then newly allocated. It is the callers responsibility to free the returned addresses in this way.

auth_con_setports

krb5_error_code
krb5_auth_con_setports(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_address * `local_port`,
krb5_address * `remote_port`)

Copies the `local_port` and `remote_port` addresses into the `auth_context`. If either address is NULL, the previous address remains in place. These addresses are set by **krb5_auth_con_genaddrs**().

krb5_error_code auth_con_setuserkey
krb5_auth_con_setuserkey(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_keyblock * `keyblock`)

This function overloads the keyblock field. It is only useful prior to a **krb5_rd_req_decode**() call for user to user authentication where the server has the key and needs to use it to decrypt the incoming request. Once decrypted this key is no longer necessary and is then overwritten with the session key sent by the client.

krb5_error_code auth_con_getkey
krb5_auth_con_getkey(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ OUT */*
krb5_keyblock ** `keyblock`)

Retrieves the keyblock stored in `auth_context`. The memory allocated in this function should be freed with a call to **krb5_free_keyblock**().

krb5_error_code auth_con_getlocalsubkey
krb5_auth_con_getlocalsubkey(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ OUT */*
krb5_keyblock ** `keyblock`)

Retrieves the local_subkey keyblock stored in `auth_context`. The memory allocated in this function should be freed with a call to **krb5_free_keyblock**().

krb5_error_code auth_con_getremotesubkey
krb5_auth_con_getremotesubkey(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ OUT */*
krb5_keyblock ** `keyblock`)

Retrieves the remote_subkey keyblock stored in `auth_context`. The memory allocated in this function should be freed with a call to **krb5_free_keyblock**().

krb5_error_code auth_setcksumtype
krb5_auth_setcksumtype(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_cksumtype `cksumtype`)

Sets the checksum type used by the other functions in the library.

auth_getlocalseqnumber

krb5_error_code
krb5_auth_getlocalseqnumber(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_int32 * `seqnumber`)

Retrieves the local sequence number that was used during authentication and stores it in `seqnumber`.

auth_getremoteseqnumber

krb5_error_code
krb5_auth_getremoteseqnumber(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_int32 * `seqnumber`)

Retrieves the remote sequence number that was used during authentication and stores it in `seqnumber`.

auth_getauthenticator

krb5_error_code
krb5_auth_getauthenticator(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ OUT */*
krb5_authenticator ** `authenticator`)

Retrieves the authenticator that was used during mutual authentication. It is the callers responsibility to free the memory allocated to `authenticator` by calling **krb5_free_authenticator**().

auth_con_initivector

krb5_error_code
krb5_auth_con_initivector(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`)

Allocates memory for and zeros the initial vector in the `auth_context` keyblock.

auth_con_setivector

krb5_error_code
krb5_auth_con_setivector(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context * `auth_context`,
/ IN */*
krb5_pointer `ivector`)

Sets the i_vector portion of `auth_context` to `ivector`.

krb5_error_code auth_con_setrcache
krb5_auth_con_setrcache(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
krb5_rcache `rcache`)

Sets the replay cache that is used by the authentication routines to `rcache`.

4.1.3 Principal access functions

Principals define a uniquely named client or server instance that participates in a network communication. The following functions allow one to create, modify and access portions of the **krb5_principal**.

Other functions found in orther portions of the manual include **krb5_sname_to_principal**(), **krb5_free_principal**(),

While it is possible to directly access the data structure in the structure, it is recommended that the functions be used.

krb5_error_code parse_name
krb5_parse_name(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const char * `name`,
/ OUT */*
krb5_principal * `principal`)

Converts a single-string representation `name` of the principal name to the multi-part principal format used in the protocols.

A single-string representation of a Kerberos name consists of one or more principal name components, separated by slashes, optionally followed by the "@" character and a realm name. If the realm name is not specified, the local realm is used.

The slash and "@" characters may be quoted (i.e., included as part of a component rather than as a component separator or realm prefix) by preceding them with a backslash ("\") character. Similarly, newline, tab, backspace, and null characters may be included in a component by using $\backslash n$, $\backslash t$, $\backslash b$ or $\backslash 0$, respectively.

The realm in a Kerberos name may not contain the slash, colon or null characters.

`*principal` will point to allocated storage which should be freed by the caller (using **krb5_free_principal**()) after use.

krb5_parse_name() returns KRB5_PARSE_MALFORMED if the string is badly formatted, or ENOMEM if space for the return value can't be allocated.

unparse_name

krb5_error_code
krb5_unparse_name(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_const_principal `principal`,
/ OUT */*
char ** `name`)

Converts the multi-part principal name `principal` from the format used in the protocols to a single-string representation of the name. The resulting single-string representation will use the format and quoting conventions described for **krb5_parse_name**() above.

`*name` points to allocated storage and should be freed by the caller when finished.

krb5_unparse_name() returns KRB_PARSE_MALFORMED if the principal does not contain at least 2 components, and system errors (ENOMEM if unable to allocate memory).

unparse_name_ext

krb5_error_code
krb5_unparse_name_ext(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_const_principal `principal`,
/ IN/OUT */*
char ** `name`,
int * `size`)

krb5_unparse_name_ext() is designed for applications which must unparse a large number of principals, and are concerned about the speed impact of needing to do a lot of memory allocations and deallocations. It functions similarly to **krb5_unparse_name**() except if `*name` is non-null, in which case, it is assumed to contain an allocated buffer of size `*size` and this buffer will be resized with **realloc**() to hold the unparsed name. Note that in this case, `size` must not be null.

If `size` is non-null (whether or not `*name` is null when the function is called), it will be filled in with the size of the unparsed name upon successful return.

princ_realm

krb5_data
krb5_princ_realm(*/* IN/OUT */*
`krb5_context`context `krb5_principal`principal)

A macro which returns the realm of `principal`.

princ_set_realm

void
krb5_princ_set_realm(*/* IN/OUT */*
`krb5_context`context `krb5_principal`principal `krb5_data *`realm)

A macro which returns sets the realm of `principal` to `realm`.

princ_set_realm_data

void
krb5_princ_set_realm_data(*/* IN/OUT */*
`krb5_context`context `krb5_principal`principal `char *`data)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

A macro which returns sets the data portion of the realm of `principal` to `data`.

void princ_set_realm_length
krb5_princ_set_realm_length(*/* IN/OUT */*
`krb5_context`context `krb5_principal`principal `int`length)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

A macro which returns sets the length `principal` to `length`.

void princ_size
krb5_princ_size(*/* IN/OUT */*
`krb5_context`context `krb5_principal`principal)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

A macro which gives the number of elements in the principal. May also be used on the left size of an assignment.

void princ_type
krb5_princ_type(*/* IN/OUT */*
`krb5_context`context `krb5_principal`principal)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

A macro which gives the type of the principal. May also be used on the left size of an assignment.

princ_data
krb5_princ_data(`krb5_context`context `krb5_principal`principal)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

A macro which gives the pointer to data portion of the principal. May also be used on the left size of an assignment.

princ_component
krb5_princ_component(`krb5_context`context `krb5_principal`principal `int`i)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

A macro which gives the pointer to ith element of the principal. May also be used on the left size of an assignment.

build_principal

krb5_error_code
krb5_build_principal(*/* IN/OUT */*
krb5_context `context`,
/ OUT */*
krb5_principal * `princ`,
/ IN */*
int `rlen`,
const char * `realm`,
char `*s1, *s2, ..., 0`)

build_principal_va

krb5_error_code
krb5_build_principal_va(*/* IN/OUT */*
krb5_context `context`,
/ OUT */*
krb5_principal * `princ`,
/ IN */*
int `rlen`,
const char * `realm`,
va_list `ap`)

krb5_build_principal() and **krb5_build_principal_va**() perform the same function; the former takes variadic arguments, while the latter takes a pre-computed varargs pointer.

Both functions take a realm name `realm`, realm name length `rlen`, and a list of null-terminated strings, and fill in a pointer to a principal structure `princ`, making it point to a structure representing the named principal. The last string must be followed in the argument list by a null pointer.

build_principal_ext

krb5_error_code
krb5_build_principal_ext(*/* IN/OUT */*
krb5_context `context`,
/ OUT */*
krb5_principal * `princ`,
/ IN */*
int `rlen`,
const char * `realm`,
`int len1, char *s1, int len2, char *s2, ..., 0`)

krb5_build_principal_ext() is similar to **krb5_build_principal**() but it takes its components as a list of (length, contents) pairs rather than a list of null-terminated strings. A length of zero indicates the end of the list.

copy_principal

krb5_error_code
krb5_copy_principal(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_const_principal `inprinc`,
/ OUT */*
krb5_principal * `outprinc`)

Copy a principal structure, filling in `*outprinc` to point to the newly allocated copy, which should be freed with **krb5_free_principal**().

krb5_boolean principal_compare
krb5_principal_compare(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_const_principal `princ1`,
krb5_const_principal `princ2`)

If the two principals are the same, return TRUE, else return FALSE.

krb5_boolean realm_compare
krb5_realm_compare(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_const_principal `princ1`,
krb5_const_principal `princ2`)

If the realms of the two principals are the same, return TRUE, else return FALSE.

krb5_error_code 425_conv_principal
krb5_425_conv_principal(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const char * `name`,
const char * `instance`,
const char * `realm`,
/ OUT */*
krb5_principal * `princ`)

Build a principal `princ` from a V4 specification made up of `name.instance@realm`. The routine is site-customized to convert the V4 naming scheme to a V5 one. For instance, the V4 "rcmd" is changed to "host".

The returned principal should be freed with **krb5_free_principal**().

4.1.4 The application functions

krb5_error_code encode_kdc_rep
krb5_encode_kdc_rep(*/* IN */*
const krb5_msgtype `type`,
const krb5_enc_kdc_rep_part * `encpart`,
krb5_encrypt_block * `eblock`,
const krb5_keyblock * `client_key`,
/ IN/OUT */*
krb5_kdc_rep * `dec_rep`,
/ OUT */*
krb5_data ** `enc_rep`)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Takes KDC rep parts in *rep and *encpart, and formats it into *enc_rep, using message type type and encryption key client_key and encryption block eblock.

enc_rep->data will point to allocated storage upon non-error return; the caller should free it when finished.

Returns system errors.

decode_kdc_rep

krb5_error_code
krb5_decode_kdc_rep(*/* IN/OUT */*
krb5_context context,
/ IN */*
krb5_data * enc_rep,
const krb5_keyblock * key,
const krb5_enctype etype,
/ OUT */*
krb5_kdc_rep ** dec_rep)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Takes a KDC_REP message and decrypts encrypted part using etype and *key, putting result in *dec_rep. The pointers in dec_rep are all set to allocated storage which should be freed by the caller when finished with the response (by using **krb5_free_kdc_rep**()).

If the response isn't a KDC_REP (tgs or as), it returns an error from the decoding routines.

Returns errors from encryption routines, system errors.

kdc_rep_decrypt_proc

krb5_error_code
krb5_kdc_rep_decrypt_proc(*/* IN/OUT */*
krb5_context context,
/ IN */*
const krb5_keyblock * key,
krb5_const_pointer decryptarg,
/ IN/OUT */*
krb5_kdc_rep * dec_rep)

Decrypt the encrypted portion of dec_rep, using the encryption key key. The parameter decryptarg is ignored.

The result is in allocated storage pointed to by dec_rep->enc_part2, unless some error occurs.

This function is suitable for use as the decrypt_proc argument to **krb5_get_in_tkt**().

krb5_error_code encrypt_tkt_part
krb5_encrypt_tkt_part(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_keyblock * `srv_key`,
/ IN/OUT */*
krb5_ticket * `dec_ticket`)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Encrypts the unecrypted part of the ticket found in `dec_ticket->enc_part2` using `srv_key`, and places result in `dec_ticket->enc_part`. The `dec_ticket->enc_part` will be allocated by this function.

Returns errors from encryption routines, system errors

`enc_part->data` is allocated and filled in with encrypted stuff.

krb5_error_code decrypt_tkt_part
krb5_decrypt_tkt_part(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_keyblock * `srv_key`,
/ IN/OUT */*
krb5_ticket * `dec_ticket`)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Takes encrypted `dec_ticket->enc_part`, decrypts with `dec_ticket->etype` using `srv_key`, and places result in `dec_ticket->enc_part2`. The storage of `dec_ticket->enc_part2` will be allocated before return.

Returns errors from encryption routines, system errors

krb5_error_code send_tgs
krb5_send_tgs(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_flags `kdcoptions`,
const krb5_ticket_times * `timestruct`,
const krb5_enctype * `etypes`,
const krb5_cksumtype `sumtype`,
krb5_const_principal `sname`,
krb5_address * const * `addrs`,
krb5_authdata * const * `authorization_data`,
krb5_pa_data * const * `padata`,
const krb5_data * `second_ticket`,
/ IN/OUT */*
krb5_creds * `in_cred`,
/ OUT */*
krb5_response * `rep`)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Sends a request to the TGS and waits for a response. `kdcoptions` is used for the options in the KRB_TGS_REQ. `timestruct` values are used for from, till, and rtime in the KRB_TGS_REQ. `etypes` is a list of etypes used in the KRB_TGS_REQ. `sumtype` is used for the checksum in the AP_REQ in the KRB_TGS_REQ. `sname` is used for sname in the KRB_TGS_REQ. `addrs`, if non-NULL, is used for addresses in the KRB_TGS_REQ. `authorization_data`, if non-NULL, is used for `authorization_data` in the KRB_TGS_REQ. `padata`, if non-NULL, is combined with any other supplied pre-authentication data for the KRB_TGS_REQ. `second_ticket`, if required by options, is used for the 2nd ticket in the KRB_TGS_REQ. `in_cred` is used for the ticket and session key in the KRB_AP_REQ header in the KRB_TGS_REQ.

The KDC realm is extracted from `in_cred->server`'s realm.

The response is placed into `*rep`. `rep->response.data` is set to point at allocated storage which should be freed by the caller when finished.

Returns system errors.

get_cred_from_kdc

krb5_error_code
krb5_get_cred_from_kdc(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_ccache `ccache`,
krb5_creds * `in_cred`,
/ OUT */*
krb5_cred ** `out_cred`,
krb5_creds *** `tgts`)

Retrieve credentials for principal `in_cred->client`, server `creds->server`, possibly `creds->second_ticket` if needed by the ticket flags.

`ccache` is used to fetch initial TGT's to start the authentication path to the server.

Credentials are requested from the KDC for the server's realm. Any TGT credentials obtained in the process of contacting the KDC are returned in an array of credentials; `tgts` is filled in to point to an array of pointers to credential structures (if no TGT's were used, the pointer is zeroed). TGT's may be returned even if no useful end ticket was obtained.

The returned credentials are NOT cached.

If credentials are obtained, `creds` is filled in with the results; `creds->ticket` and `creds->keyblock->key` are set to allocated storage, which should be freed by the caller when finished.

Returns errors, system errors.

krb5_error_code get_cred_via_tkt
krb5_get_cred_via_tkt(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_creds * `tkt`,
const krb5_flags `kdcoptions`,
krb5_address *const * `address`,
krb5_creds * `in_cred`,
/ OUT */*
krb5_creds ** `out_cred`)

Takes a ticket `tkt` and a target credential `in_cred`, attempts to fetch a TGS from the KDC. Upon success the resulting is stored in `out_cred`. The memory allocated in `out_cred` should be freed by the called when finished by using **krb5_free_creds**().

`kdcoptions` refers to the options as listed in Table 2. The optional `address` is used for addressed in the KRB_TGS_REQ (see **krb5_send_tgs**()).

Returns errors, system errors.

krb5_error_code get_credentials
krb5_get_credentials(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_flags `options`,
krb5_ccache `ccache`,
krb5_creds * `in_creds`,
/ OUT */*
krb5_creds * `out_creds`)

This routine attempts to use the credentials cache `ccache` or a TGS exchange to get an additional ticket for the client identified by `in_creds->client`, with following information:

- **The server** identified by `in_creds->server`
- **The options** in `options`. Valid choices are KRB5_GC_USER_USER and KRB5_GC_GC_CACHED
- **The expiration date** specified in `in_creds->times.endtime`
- **The session key type** specified in `in_creds->keyblock.keytype` if it is non-zero.

If `options` specifies KRB5_GC_CACHED, then **krb5_get_credentials**() will only search the credentials cache for a ticket.

If `options` specifies KRB5_GC_USER_USER, then **krb5_get_credentials**() will get credentials for a user to user authentication. In a user to user authentication, the secret key for the server is the session key from the server's ticket-granting-ticket (TGT). The TGT is passed from the server to the client over the network — this is safe since the TGT is encrypted in a key known only by the Kerberos server — and the client must pass this TGT to **krb5_get_credentials**() in `in_creds->second_ticket`. The Kerberos server will use this TGT to construct a user to user ticket which can be verified by the server by using the session key from its TGT.

The effective **expiration date** is the minimum of the following:

- The expiration date as specified in `in_creds->times.endtime`
- The requested start time plus the maximum lifetime of the server as specified by the server's entry in the Kerberos database.
- The requested start time plus the maximum lifetime of tickets allowed in the local site, as specified by the KDC. This is currently a compile-time option, KRB5_KDB_MAX_LIFE in config.h, and is by default 1 day.

If any special authorization data needs to be included in the ticket, — for example, restrictions on how the ticket can be used — they should be specified in `in_creds->authdata`. If there is no special authorization data to be passed, `in_creds->authdata` should be NULL.

Any returned ticket and intermediate ticket-granting tickets are stored in `ccache`.

Returns errors from encryption routines, system errors.

get_in_tkt

```
krb5_error_code
krb5_get_in_tkt(/* IN/OUT */
        krb5_context context,
        /* IN */
        const krb5_flags options,
        krb5_address * const * addrs,
        const krb5_enctype * etypes,
        const krb5_preauthtype * ptypes,
        krb5_error_code (*key_proc)(krb5_context context,
                                    const krb5_keytype type,
                                    krb5_data * salt,
                                    krb5_const_pointer keyseed,
                                    krb5_keyblock ** key),
        krb5_const_pointer keyseed,
        krb5_error_code (*decrypt_proc)(krb5_context context,
                                        const krb5_keyblock * key,
                                        krb5_const_pointer decryptarg,
                                        krb5_kdc_rep * dec_rep),
        krb5_const_pointer decryptarg,
        /* IN/OUT */
        krb5_creds * creds,
        krb5_ccache ccache,
        krb5_kdc_rep ** ret_as_reply)
```

This all-purpose initial ticket routine, usually called via **krb5_get_in_tkt_with_skey**() or **krb5_get_in_tkt_with_password**() or **krb5_get_in_tkt_with_keytab**().

Attempts to get an initial ticket for `creds->client` to use server `creds->server`, using the following: the realm from `creds->client`; the options in `options` (listed in Table 2); and `ptypes`, the preauthentication method (valid preauthentication methods are listed in Table 2). **krb5_get_in_tkt**() requests encryption type `etypes` (valid encryption types are ETYPE_DES_CBC_CRC and ETYPE_RAW_DES_CBC), using `creds->times.starttime`, `creds->times.endtime`, `creds->times.renew_till` as from, till, and rtime. `creds->times.renew_till` is ignored unless the RENEWABLE option is requested.

key_proc is called, with context, keytype, keyseed andpadata as arguments, to fill in key to be used for decryption. The valid key types for keytype are KEYTYPE_NULL,[9] and KEYTYPE_DES.[10] However, KEYTYPE_DES is the only key type supported by MIT kerberos. The content of keyseed depends on the key_proc being used. The padata passed to key_proc is the preauthentication data returned by the KDC as part of the reply to the initial ticket request. It may contain an element of type KRB5_PADATA_PW_SALT, which key_proc should use to determine what salt to use when generating the key. key_proc should fill in key with a key for the client, or return an error code.

decrypt_proc is called to perform the decryption of the response (the encrypted part is in dec_rep->enc_part; the decrypted part should be allocated and filled into dec_rep->enc_part2. decryptarg is passed on to decrypt_proc, and its content depends on the decrypt_proc being used.

If addrs is non-NULL, it is used for the addresses requested. If it is null, the system standard addresses are used.

If ret_as_reply is non-NULL, it is filled in with a pointer to a structure containing the reply packet from the KDC. Some programs may find it useful to have direct access to this information. For example, it can be used to obtain the pre-authentication data passed back from the KDC. The caller is responsible for freeing this structure by using **krb5_free_kdc_rep**().

If etypes is non-NULL, the it is used as for the list of valid encyrption types. Otherwise, the context default is used (as returned by **krb5_get_default_in_tkt_etypes**().

A succesful call will place the ticket in the credentials cache ccache and fill in creds with the ticket information used/returned.

Returns system errors, preauthentication errors, encryption errors.

krb5_error_code get_in_tkt_with_password
krb5_get_in_tkt_with_password(*/* IN/OUT */*
krb5_context context,
/ IN */*
const krb5_flags options,
krb5_address * const * addrs,
const krb5_enctype * etypes,
const krb5_preauthtype * pre_auth_types,
const char * password,
krb5_ccache ccache,
/ IN/OUT */*
krb5_creds * creds,
krb5_kdc_rep ** ret_as_reply)

Attempts to get an initial ticket using the null-terminated string password. If password is NULL, the password is read from the terminal using as a prompt the globalname krb5_default_pwd_prompt1.

The password is converted into a key using the appropriate string-to-key conversion function for the specified keytype, and using any salt data returned by the KDC in response to the authentication request.

[9] See RFC section 6.3.1
[10] See RFC section 6.3.4

See **krb5_get_in_tkt**() for documentation of the `options`, `addrs`, `pre_auth_type`, `etype`, `keytype`, `ccache`, `creds` and `ret_as_reply` arguments.

Returns system errors, preauthentication errors, encryption errors.

get_in_tkt_with_keytab

krb5_error_code
krb5_get_in_tkt_with_keytab(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_flags `options`,
krb5_address * const * `addrs`,
const krb5_enctype * `etypes`,
const krb5_preauthtype * `pre_auth_types`,
const krb5_keytab * `keytab`,
krb5_ccache `ccache`,
/ IN/OUT */*
krb5_creds * `creds`,
krb5_kdc_rep ** `ret_as_reply`)

Attempts to get an initial ticket using `keytab`. If `keytab` is NULL, the default keytab is used (e.g., */etc/v5srvtab*).

See **krb5_get_in_tkt**() for documentation of the `options`, `addrs`, `pre_auth_type`, `etype`, `ccache`, `creds` and `ret_as_reply` arguments.

Returns system errors, preauthentication errors, encryption errors.

get_in_tkt_with_skey

krb5_error_code
krb5_get_in_tkt_with_skey(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_flags `options`,
krb5_address * const * `addrs`,
const krb5_enctype * `etypes`,
const krb5_preauthtype * `pre_auth_types`,
const krb5_keyblock * `key`,
krb5_ccache `ccache`,
/ IN/OUT */*
krb5_creds * `creds`,
krb5_kdc_rep ** `ret_as_reply`)

Attempts to get an initial ticket using `key`. If `key` is NULL, an appropriate key is retrieved from the system key store (e.g., */etc/v5srvtab*).

See **krb5_get_in_tkt**() for documentation of the `options`, `addrs`, `pre_auth_type`, `etype`, `ccache`, `creds` and `ret_as_reply` arguments.

Returns system errors, preauthentication errors, encryption errors.

krb5_error_code mk_req
krb5_mk_req(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context * `auth_context`,
/ IN */*
const krb5_flags `ap_req_options`,
char * `service`,
char * `hostname`,
krb5_data * `in_data`,
/ IN/OUT */*
krb5_ccache `ccache`,
/ OUT */*
krb5_data * `outbuf`)

Formats a KRB_AP_REQ message into `outbuf`.

The server to receive the message is specified by `hostname`. The principal of the server to receive the message is specified by `hostname` and `service`. If credentials are not present in the credentials cache `ccache` for this server, the TGS request with default parameters is used in an attempt to obtain such credentials, and they are stored in `ccache`.

`ap_req_options` specifies the KRB_AP_REQ options desired. Valid options are:

AP_OPTS_USE_SESSION_KEY
AP_OPTS_MUTUAL_REQUIRED

The checksum method to be used is as specified in `auth_context`.

`outbuf` should point to an existing **krb5_data** structure. `outbuf->length` and `outbuf->data` will be filled in on success, and the latter should be freed by the caller when it is no longer needed; if an error is returned, however, no storage is allocated and `outbuf->data` does not need to be freed.

Returns system errors, error getting credentials for `server`.

krb5_error_code mk_req_extended
krb5_mk_req_extended(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context * `auth_context`,
/ IN */*
const krb5_flags `ap_req_options`,
krb5_data * `in_data`,
krb5_creds * `in_creds`,
/ OUT */*
krb5_data * `outbuf`)

Formats a KRB_AP_REQ message into `outbuf`, with more complete options than **krb5_mk_req**().

`outbuf`, `ap_req_options`, `auth_context`, and `ccache` are used in the same fashion as for **krb5_mk_req**().

`in_creds` is used to supply the credentials (ticket and session key) needed to form the request.

If `in_creds->ticket` has no data (length == 0), then an error is returned.

During this call, the structure elements in `in_creds` may be freed and reallocated. Hence all of the structure elements which are pointers should point to allocated memory, and there should be no other pointers aliased to the same memory, since it may be deallocated during this procedure call.

If `ap_req_options` specifies AP_OPTS_USE_SUBKEY, then a subkey will be generated if need be by **krb5_generate_subkey**().

A copy of the authenticator will be stored in the `auth_context`, with the principal and checksum fields nulled out, unless an error is returned. (This is to prevent pointer sharing problems; the caller shouldn't need these fields anyway, since the caller supplied them.)

Returns system errors, errors contacting the KDC, KDC errors getting a new ticket for the authenticator.

generate_subkey

krb5_error_code
krb5_generate_subkey(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_keyblock * `key`,
/ OUT */*
krb5_keyblock ** `subkey`)

Generates a pseudo-random sub-session key using the encryption system's random key functions, based on the input `key`.

`subkey` is filled in to point to the generated subkey, unless an error is returned. The returned key (i.e., `*subkey`) is allocated and should be freed by the caller with **krb5_free_keyblock**() when it is no longer needed.

rd_req

krb5_error_code
krb5_rd_req(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context * `auth_context`,
/ IN */*
const krb5_data * `inbuf`,
krb5_const_principal `server`,
krb5_keytab `keytab`,
/ IN/OUT */*
krb5_flags * `ap_req_options`,
/ OUT */*
krb5_ticket ** `ticket`)

Parses a KRB_AP_REQ message, returning its contents. Upon successful return, if `ticket` is non-NULL, `*ticket` will be modified to point to allocated storage containing the ticket information. The caller is responsible for deallocating this space by using **krb5_free_ticket**().

`inbuf` should contain the KRB_AP_REQ message to be parsed.

If `auth_context` is NULL, one will be generated and freed internally by the function.

`server` specifies the expected server's name for the ticket. If `server` is NULL, then any server name will be accepted if the appropriate key can be found, and the caller should verify that the server principal matches some trust criterion.

If `server` is not NULL, and a replay detaction cache has not been established with the `auth_context`, one will be generated.

`keytab` specifies a keytab containing generate a decryption key. If NULL, `krb5_kt_default` will be used to find the default keytab and the key taken from there[11].

If a keyblock is present in the `auth_context`, it will be used to decrypt the ticket request and the keyblock freed with **krb5_free_keyblock**(). This is useful for user to user authentication. If no keyblock is specified, the `keytab` is consulted for an entry matching the requested keytype, server and version number and used instead.

The authentcator in the request is decrypted and stored in the `auth_context`. The client specified in the decrypted authenticator is compared to the client specified in the decoded ticket to ensure that the compare.

If the remote_addr portion of the `auth_context` is set, then this routine checks if the request came from the right client.

`sender_addr` specifies the address(es) expected to be present in the ticket.

The replay cache is checked to see if the ticket and authenticator have been seen and if so, returns an error. If not, the ticket and authenticator are entered into the cache.

Various other checks are made of the decoded data, including, cross-realm policy, clockskew and ticket validation times.

The keyblock, subkey, and sequence number of the request are all stored in the `auth_context` for future use.

If the request has the AP_OPTS_MUTUAL_REQUIRED bit set, the local sequence number, which is stored in the auth_context, is XORed with the remote sequence number in the request.

If `ap_req_options` is non-NULL, it will be set to contain the application request flags.

Returns system errors, encryption errors, replay errors.

krb5_error_code rd_req_decoded
krb5_rd_req_decoded(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context * `auth_context`,
/ IN */*
const krb5_ap_req * `req`,
krb5_const_principal `server`,
/ IN/OUT */*
krb5_keytab `keytab`,
/ OUT */*
krb5_ticket ** `ticket`)

Essentially the same as **krb5_rd_req**(), but uses a decoded AP_REQ as the input rather than an encoded input.

[11]i.e., srvtab file in Kerberos V4 parlance

mk_rep

krb5_error_code
krb5_mk_rep(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ OUT */*
krb5_data * `outbuf`)

Formats and encrypts an AP_REP message, including in it the data in the authentp portion of `auth_context`, encrypted using the keyblock portion of `auth_context`.

When successfull, `outbuf->length` and `outbuf->data` are filled in with the length of the AP_REQ message and allocated data holding it. `outbuf->data` should be freed by the caller when it is no longer needed.

If the flags in `auth_context` indicate that a sequence number should be used (either KRB5_AUTH_CONTEXT_DO_SEQUENCE or KRB5_AUTH_CONTEXT_RET_SEQUENCE) and the local sequqnce number in the `auth_context` is 0, a new number will be generated with **krb5_generate_seq_number**().

Returns system errors.

rd_rep

krb5_error_code
krb5_rd_rep(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
const krb5_data * `inbuf`,
/ OUT */*
krb5_ap_rep_enc_part ** `repl`)

Parses and decrypts an AP_REP message from `*inbuf`, filling in `*repl` with a pointer to allocated storage containing the values from the message. The caller is responsible for freeing this structure with **krb5_free_ap_rep_enc_part**().

The keyblock stored in `auth_context` is used to decrypt the message after establishing any key pre-processing with **krb5_process_key**().

Returns system errors, encryption errors, replay errors.

mk_error

krb5_error_code
krb5_mk_error(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_error * `dec_err`,
/ OUT */*
krb5_data * `enc_err`)

Formats the error structure `*dec_err` into an error buffer `*enc_err`.

The error buffer storage (`enc_err->data`) is allocated, and should be freed by the caller when finished.

Returns system errors.

krb5_error_code rd_error
krb5_rd_error(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_data * `enc_errbuf`,
/ OUT */*
krb5_error ** `dec_error`)

Parses an error protocol message from `enc_errbuf` and fills in `*dec_error` with a pointer to allocated storage containing the error message. The caller is reponsible for freeing this structure by using **krb5_free_error**().

Returns system errors.

krb5_error_code generate_seq_number
krb5_generate_seq_number(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_keyblock * `key`,
/ OUT */*
krb5_int32 * `seqno`)

Generates a pseudo-random sequence number suitable for use as an initial sequence number for the KRB_SAFE and KRB_PRIV message processing routines.

`key` parameterizes the choice of the random sequence number, which is filled into `*seqno` upon return.

krb5_error_code sendauth
krb5_sendauth(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context * `auth_context`,
/ IN */*
krb5_pointer `fd`,
char * `appl_version`,
krb5_principal `client`,
krb5_principal `server`,
krb5_flags `ap_req_options`,
krb5_data * `in_data`,
krb5_creds * `in_creds`,
/ IN/OUT */*
krb5_ccache `ccache`,
/ OUT */*
krb5_error ** `error`,
krb5_ap_rep_enc_part ** `rep_result`,
krb5_creds ** `out_creds`)

krb5_sendauth() provides a convenient means for client and server programs to send authenticated messages to one another through network connections. **krb5_sendauth**() sends an authenticated ticket from the client program to the server program using the network connection specified by `fd`. In the MIT Unix implementation, `fd` should be a pointer to a file descriptor describing the network socket. This can be changed in other implementations, however, if the routines **krb5_read_message**(), **krb5_write_message**(), **krb5_net_read**(), and **krb5_net_write**() are changed.

The paramter `appl_version` is a string describing the application protocol version which the client is expecting to use for this exchange. If the server is using a different application protocol, an error will be returned.

The parameters `client` and `server` specify the kerberos principals for the client and the server. They are ignored if `in_creds` is non-null. Otherwise, `server` must be non-null, but `client` may be null, in which case the client principal used is the one in the credential cache's default principal.

The `ap_req_options` parameters specifies the options which should be passed to **krb5_mk_req**(). Valid options are listed in Table 4.1.4. If `ap_req_options` specifies MUTUAL_REQUIRED, then **krb5_sendauth**() will perform a mutual authentication exchange, and if `rep_result` is non-null, it will be filled in with the result of the mutual authentication exchange; the caller should free `*rep_result` with **krb5_free_ap_rep_enc_part**() when done with it.

If `in_creds` is non-null, then `in_creds->client` and `in_creds->server` must be filled in, and either the other structure fields should be filled in with valid credentials, or `in_creds->ticket.length` should be zero. If `in_creds->ticket.length` is non-zero, then `in_creds` will be used as-is as the credentials to send to the server, and `ccache` is ignored; otherwise, `ccache` is used as described below, and `out_creds` , if not NULL, is filled in with the retrieved credentials.

ccache() specifies the credential cache to use when one is needed (i.e., when **in_creds**() is null or `in_creds->ticket.length` is zero). When a credential cache is not needed, **ccache**() is ignored. When a credential cache is needed and **ccache**() is null, the default credential cache is used. Note that if the credential cache is needed and does not contain the needed credentials, they will be retrieved from the KDC and stored in the credential cache.

If mutual authentication is used and `rep_result` is non-null, the sequence number for the server is available to the caller in `*rep_result->seq_number`. (If mutual authentication is not used, there is no way to negotiate a sequence number for the server.)

If an error occurs during the authenticated ticket exchange and `error` is non-null, the error packet (if any) that was sent from the server will be placed in it. This error should be freed with **krb5_free_error**().

recvauth

krb5_error_code
krb5_recvauth(*/* IN/OUT */*
    krb5_context `context`,
    krb5_auth_context * `auth_context`,
    / IN */*
    krb5_pointer `fd`,
    char * `appl_version`,
    krb5_principal `server`,
    char * `rc_type`,
    krb5_int32 `flags`,
    krb5_keytab `keytab`,
    / OUT */*
    krb5_ticket ** `ticket`)

krb5_recvauth() provides a convenient means for client and server programs to send authenticated messages to one another through network connections. **krb5_sendauth**() is the matching routine to **krb5_recvauth**() for the server. **krb5_recvauth**()

will engage in an authentication dialogue with the client program running **krb5_sendauth**() to authenticate the client to the server. In addition, if requested by the client, **krb5_recvauth**() will provide mutual authentication to prove to the client that the server represented by **krb5_recvauth**() is legitimate.

`fd` is a pointer to the network connection. As in **krb5_sendauth**(), in the MIT Unix implementation `fd` is a pointer to a file descriptor.

The parameter `appl_version` is a string describing the application protocol version which the server is expecting to use for this exchange. If the client is using a different application protocol, an error will be returned and the authentication exchange will be aborted.

If `server` is non-null, then **krb5_recvauth**() verifies that the server principal requested by the client matches `server`. If not, an error will be returned and the authentication exchange will be aborted.

The parameters `server`, `auth_context`, and `keytab` are used by **krb5_rd_req**() to obtain the server's private key.

If `server` is non-null, the princicpal component of it is ysed to determine the replay cache to use. Otherwise, **krb5_recvauth**() will use a default replay cache.

The `flags` argument allows the caller to modify the behavior of **krb5_recvauth**(). For non-library callers, `flags` should be 0.

`ticket` is optional and is only filled in if non-null. It is filled with the data from the ticket sent by the client, and should be freed with **krb5_free_ticket**() when it is no longer needed.

krb5_error_code mk_safe
krb5_mk_safe(*/* IN/OUT */*
    krb5_context `context`,
    krb5_auth_context `auth_context`,
    / IN */*
    const krb5_data * `userdata`,
    / OUT */*
    krb5_data * `outbuf`,
    / IN/OUT */*
    krb5_replay_data * `outdata`)

Formats a KRB_SAFE message into `outbuf`.

`userdata` is formatted as the user data in the message. Portions of `auth_context` specify the checksum type; the keyblockm which might be used to seed the checksum; full addresses (host and port) for the sender and receiver. The `local_addr` portion of `*auth_context` is used to form the addresses usedin the KRB_SAFE message. The `remote_addr` is optional; if the receiver's address is not known, it may be replaced by NULL. `local_addr`, however, is mandatory.

The `auth_context` flags select whether sequence numbers or timestamps should be used to identify the message. Valid flags are listed below.

Symbol	Meaning
KRB5_AUTH_CONTEXT_DO_TIME	Use timestamps and replay cache
KRB5_AUTH_CONTEXT_RET_TIME	Copy timestamp to `*outdata`
KRB5_AUTH_CONTEXT_DO_SEQUENCE	Use sequence numbers
KRB5_AUTH_CONTEXT_RET_SEQUENCE	Copy sequence numbers to `*outdata`

If timestamps are to be used (i.e., if KRB5_AUTH_CONTEXT_DO_TIME is set), an entry describing the message will be entered in the replay cache so that the caller may detect if this message is sent back to him by an attacker. If KRB5_AUTH_CONTEXT_DO_TIME_NOTIME is not set, the `auth_context` replay cache is not used.

If sequence numbers are to be used (i.e., if either KRB5_AUTH_CONTEXT_DO_SEQUENCE or KRB5_AUTH_CONTEXT_RET_SEQUENEC is set), then `auth_context` local sequence number will be placed in the protected message as its sequence number.

The `outbuf` buffer storage (i.e., `outbuf->data`) is allocated, and should be freed by the caller when finished.

Returns system errors, encryption errors.

rd_safe

krb5_error_code
krb5_rd_safe(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
const krb5_data * `inbuf`,
/ OUT */*
krb5_data * `outbuf`,
/ IN/OUT */*
krb5_replay_data * `outdata`)

Parses a KRB_SAFE message from `inbuf`, placing the data in `*outbuf` after verifying its integrity.

The keyblock used for verifying the integrity of the message is taken from the `auth_context` local_subkey, remote_subkey, or keyblock. The keyblock is chosen in the above order by the first one which is not NULL.

The remote_addr and localaddr portions of the `*auth_context` specify the full addresses (host and port) of the sender and receiver, and must be of type **ADDRTYPE_ADDRPORT**.

The `remote_addr` parameter is mandatory; it specifies the address of the sender. If the address of the sender in the message does not match `remote_addr`, the error KRB5KRB_AP_ERR_BADADDR will be returned.

If `local_addr` is non-NULL, then the address of the receiver in the message much match it. If it is null, the receiver address in the message will be checked against the list of local addresses as returned by **krb5_os_localaddr**(). If the check fails, KRB5KRB_AP_ERR_BADARRD is returned.

The `outbuf` buffer storage (i.e., `outbuf->data` is allocated storage which the caller should free when it is no longer needed.

If auth_context_flags portion of `auth_context` indicates that sequence numbers are to be used (i.e., if KRB5_AUTH_CONTEXT_DOSEQUENCE is set in it), The `remote_seq_number` portion of `auth_context` is compared to the sequence number for the message, and KRB5_KRB_AP_ERR_BADORDER is returned if it does not match. Otherwise, the sequence number is not used.

If timestamps are to be used (i.e., if KRB5_AUTH_CONTEXT_DO_TIME is set in the `auth_context`), then two additional checks are performed:

- The timestamp in the message must be within the permitted clock skew (which is usually five minutes), or KRB5KRB_AP_ERR_SKEW is returned.
- The message must not be a replayed message, according to `rcache`.

Returns system errors, integrity errors.

krb5_error_code mk_priv
krb5_mk_priv(*/* IN/OUT */*
krb5_context `context`,
krb5_auth_context `auth_context`,
/ IN */*
const krb5_data * `userdata`,
/ OUT */*
krb5_data * `outbuf`,
krb5_replay_data * `outdata`)

Formats a KRB_PRIV message into `outbuf`. Behaves similarly to **krb5_mk_safe**(), but the message is encrypted and integrity-protected rather than just integrity-protected.

`inbuf`, `auth_context`, `outdata` and `outbuf` function as in **krb5_mk_safe**().

As in **krb5_mk_safe**(), the remote_addr and remote_port part of the `auth_context` is optional; if the receiver's address is not known, it may be replaced by NULL. The local_addr, however, is mandatory.

The encryption type is taken from the `auth_context` keyblock portion. If i_vector portion of the `auth_context` is non-null, it is used as an initialization vector for the encryption (if the chosen encryption type supports initialization vectors) and its contents are replaced with the last block of encrypted data upon return.

The flags from the `auth_context` selects whether sequence numbers or timestamps should be used to identify the message. Valid flags are listed below.

Symbol	Meaning
KRB5_AUTH_CONTEXT_DO_TIME	Use timestamps in replay cache
KRB5_AUTH_CONTEXT_RET_TIME	Use timestamps in output data
KRB5_AUTH_CONTEXT_DO_SEQUENCE	Use sequence numbers in replay cache
KRB5_AUTH_CONTEXT_RET_SEQUENCE	Use sequence numbers in replay cache and output data

Returns system errors, encryption errors.

rd_priv

```
krb5_error_code
krb5_rd_priv(/* IN/OUT */
             krb5_context context,
             krb5_auth_context auth_context,
             /* IN */
             const krb5_data * inbuf,
             /* OUT */
             krb5_data * outbuf,
             krb5_data * outdata)
```

Parses a KRB_PRIV message from `inbuf`, placing the data in `*outbuf` after decrypting it. Behaves similarly to **krb5_rd_safe**(), but the message is decrypted rather than integrity-checked.

`inbuf`, `auth_context`, `outdata` and `outbuf` function as in **krb5_rd_safe**().

The remote_addr part of the `auth_context` as set by **krb5_auth_con_setaddrs**() is mandatory; it specifies the address of the sender. If the address of the sender in the message does not match the remote_addr, the error KRB5KRB_AP_ERR_BADADDR will be returned.

If local_addr portion of the auth_context is non-NULL, then the address of the receiver in the message much match it. If it is null, the receiver address in the message will be checked against the list of local addresses as returned by **krb5_os_localaddr**().

The `keyblock` portion of `auth_context` specifies the key to be used for decryption of the message. If the `i_vector` element, is non-null, it is used as an initialization vector for the decryption (if the encryption type of the message supports initialization vectors) and its contents are replaced with the last block of encrypted data in the message.

The `auth_context` flags specify whether timestamps (KRB5_AUTH_CONTEXT_DO_TIME) and sequence numbers (KRB5_AUTH_CONTEXT_DO_SEQUENCE) are to be used.

Returns system errors, integrity errors.

4.1.5 Miscellaneous main functions

address_search

```
krb5_boolean
krb5_address_search(/* IN/OUT */
                    krb5_context context,
                    /* IN */
                    const krb5_address * addr,
                    krb5_address * const * addrlist)
```

If `addr` is listed in `addrlist`, or `addrlist` is null, return TRUE. If not listed, return FALSE.

address_compare

```
krb5_boolean
krb5_address_compare(/* IN/OUT */
                     krb5_context context,
                     /* IN */
                     const krb5_address * addr1,
                     const krb5_address * addr2)
```

If the two addresses are the same, return TRUE, else return FALSE.

int fulladdr_order
krb5_fulladdr_order(/* IN/OUT */
krb5_context `context`,
/* IN */
const krb5_fulladdr * `addr1`,
const krb5_fulladdr * `addr2`)

Return an ordering on the two full addresses: 0 if the same, < 0 if first is less than 2nd, > 0 if first is greater than 2nd.

int address_order
krb5_address_order(/* IN/OUT */
krb5_context `context`,
/* IN */
const krb5_address * `addr1`,
const krb5_address * `addr2`)

Return an ordering on the two addresses: 0 if the same, < 0 if first is less than 2nd, > 0 if first is greater than 2nd.

krb5_error_code copy_addresses
krb5_copy_addresses(/* IN/OUT */
krb5_context `context`,
/* IN */
krb5_address * const * `inaddr`,
/* OUT */
krb5_address *** `outaddr`)

Copy addresses in `inaddr` to `*outaddr` which is allocated memory and should be freed with **krb5_free_addresses**().

krb5_error_code copy_authdata
krb5_copy_authdata(/* IN/OUT */
krb5_context `context`,
/* IN */
krb5_authdata * const * `inauthdat`,
/* OUT */
krb5_authdata *** `outauthdat`)

Copy an authdata structure, filling in `*outauthdat` to point to the newly allocated copy, which should be freed with **krb5_free_authdata**().

krb5_error_code copy_authenticator
krb5_copy_authenticator(/* IN/OUT */
krb5_context `context`,
/* IN */
const krb5_authenticator * `authfrom`,
/* OUT */
krb5_authenticator ** `authto`)

Copy an authenticator structure, filling in `*outauthdat` to point to the newly allocated copy, which should be freed with **krb5_free_authenticator**().

copy_keyblock

krb5_error_code
krb5_copy_keyblock(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_keyblock * `from`,
/ OUT */*
krb5_keyblock ** `to`)

Copy a keyblock, filling in `*to` to point to the newly allocated copy, which should be freed with **krb5_free_keyblock**().

copy_keyblock_contents

krb5_error_code
krb5_copy_keyblock_contents(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_keyblock * `from`,
/ OUT */*
krb5_keyblock * `to`)

Copy keyblock contents from `from` to `to`, including allocated storage. The allocated storage in `to` should be freed by using **free**(`to->contents`).

copy_checksum

krb5_error_code
krb5_copy_checksum(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_checksum * `ckfrom`,
/ OUT */*
krb5_checksum ** `ckto`)

Copy a checksum structure, filling in `*ckto` to point to the newly allocated copy, which should be freed with **krb5_free_checksum**().

copy_creds

krb5_error_code
krb5_copy_creds(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_creds * `incred`,
/ OUT */*
krb5_creds ** `outcred`)

Copy a credentials structure, filling in `*outcred` to point to the newly allocated copy, which should be freed with **krb5_free_creds**().

krb5_error_code copy_data
krb5_copy_data(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_data * `indata`,
/ OUT */*
krb5_data ** `outdata`)

Copy a data structure, filling in `*outdata` to point to the newly allocated copy, which should be freed with **krb5_free_data**().

krb5_error_code copy_ticket
krb5_copy_ticket(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_ticket * `from`,
/ OUT */*
krb5_ticket ** `pto`)

Copy a ticket structure, filling in `*pto` to point to the newly allocated copy, which should be freed with **krb5_free_ticket**().

krb5_error_code get_server_rcache
krb5_get_server_rcache(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const krb5_data * `piece`,
/ OUT */*
krb5_rcache * `ret_rcache`)

Generate a replay cache name, allocate space for its handle, and open it. `piece` is used to distinguish this replay cache from others currently in use on the system. Typically, `piece` is the first component of the principal name for the client or server which is calling **krb5_get_server_rcache**().

Upon successful return, `ret_rcache` is filled in to contain a handle to an open rcache, which should be closed with **krb5_rc_close**().

4.2 Credentials cache functions

The credentials cache functions (some of which are macros which call to specific types of credentials caches) deal with storing credentials (tickets, session keys, and other identifying information) in a semi-permanent store for later use by different programs.

krb5_error_code cc_resolve
krb5_cc_resolve(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
char * `string_name`,
/ OUT */*
krb5_ccache * `id`)

Fills in `id` with a ccache identifier which corresponds to the name in `string_name`.

Requires that `string_name` be of the form "type:residual" and "type" is a type known to the library.

cc_gen_new

krb5_error_code
krb5_cc_gen_new(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_cc_ops * `ops`,
/ OUT */*
krb5_ccache * `id`)

Fills in `id` with a unique ccache identifier of a type defined by `ops`. The cache is left unopened.

cc_register

krb5_error_code
krb5_cc_register(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_cc_ops * `ops`,
krb5_boolean `override`)

Adds a new cache type identified and implemented by `ops` to the set recognized by **krb5_cc_resolve**(). If `override` is FALSE, a ticket cache type named `ops->prefix` must not be known.

cc_get_name

char *
krb5_cc_get_name(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_ccache `id`)

Returns the name of the ccache denoted by `id`.

cc_default_name

char *
krb5_cc_default_name(*/* IN/OUT */*
krb5_context `context`)

Returns the name of the default credentials cache; this may be equivalent to **getenv**(`"KRB5CCACHE"`) with an appropriate fallback.

cc_default

krb5_error_code
krb5_cc_default(*/* IN/OUT */*
krb5_context `context`,
/ OUT */*
krb5_ccache * `ccache`)

Equivalent to **krb5_cc_resolve**(**krb5_cc_default_name**(), `ccache`).

krb5_error_code cc_initialize
krb5_cc_initialize(*/* IN/OUT */*
krb5_context `context`,
krb5_ccache `id`,
/ IN */*
krb5_principal `primary_principal`)

Creates/refreshes a credentials cache identified by `id` with primary principal set to `primary_principal`. If the credentials cache already exists, its contents are destroyed.

Errors: permission errors, system errors.

Modifies: cache identified by `id`.

krb5_error_code cc_destroy
krb5_cc_destroy(*/* IN/OUT */*
krb5_context `context`,
krb5_ccache `id`)

Destroys the credentials cache identified by `id`, invalidates `id`, and releases any other resources acquired during use of the credentials cache. Requires that `id` identifies a valid credentials cache. After return, `id` must not be used unless it is first reinitialized using **krb5_cc_resolve**() or **krb5_cc_gen_new**().

Errors: permission errors.

krb5_error_code cc_close
krb5_cc_close(*/* IN/OUT */*
krb5_context `context`,
krb5_ccache `id`)

Closes the credentials cache `id`, invalidates `id`, and releases `id` and any other resources acquired during use of the credentials cache. Requires that `id` identifies a valid credentials cache. After return, `id` must not be used unless it is first reinitialized using **krb5_cc_resolve**() or **krb5_cc_gen_new**().

krb5_error_code cc_store_cred
krb5_cc_store_cred(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_ccache `id`,
krb5_creds * `creds`)

Stores `creds` in the cache `id`, tagged with `creds->client`. Requires that `id` identifies a valid credentials cache.

Errors: permission errors, storage failure errors.

cc_retrieve_cred

krb5_error_code
krb5_cc_retrieve_cred(*/* IN/OUT */*
krb5_context context,
/ IN */*
krb5_ccache id,
krb5_flags whichfields,
krb5_creds * mcreds,
/ OUT */*
krb5_creds * creds)

Searches the cache id for credentials matching mcreds. The fields which are to be matched are specified by set bits in whichfields, and always include the principal name mcreds->server. Requires that id identifies a valid credentials cache.

If at least one match is found, one of the matching credentials is returned in *creds. The credentials should be freed using **krb5_free_credentials**().

Errors: error code if no matches found.

cc_get_principal

krb5_error_code
krb5_cc_get_principal(*/* IN/OUT */*
krb5_context context,
/ IN */*
krb5_ccache id,
krb5_principal * principal)

Retrieves the primary principal of the credentials cache (as set by the **krb5_cc_initialize**() request) The primary principal is filled into *principal; the caller should release this memory by calling **krb5_free_principal**() on *principal when finished.

Requires that id identifies a valid credentials cache.

cc_start_seq_get

krb5_error_code
krb5_cc_start_seq_get(*/* IN/OUT */*
krb5_context context,
krb5_ccache id,
/ OUT */*
krb5_cc_cursor * cursor)

Prepares to sequentially read every set of cached credentials. cursor is filled in with a cursor to be used in calls to **krb5_cc_next_cred**().

cc_next_cred

krb5_error_code
krb5_cc_next_cred(*/* IN/OUT */*
krb5_context context,
krb5_ccache id,
/ OUT */*
krb5_creds * creds,
/ IN/OUT */*
krb5_cc_cursor * cursor)

Fetches the next entry from id, returning its values in *creds, and updates *cursor for the next request. Requires that id identifies a valid credentials cache and *cursor be a cursor returned by **krb5_cc_start_seq_get**() or a subsequent call to **krb5_cc_next_cred**().

Errors: error code if no more cache entries.

krb5_error_code cc_end_seq_get
krb5_cc_end_seq_get(*/* IN/OUT */*
krb5_context `context`,
krb5_ccache `id`,
krb5_cc_cursor * `cursor`)

Finishes sequential processing mode and invalidates `*cursor`. `*cursor` must never be re-used after this call.

Requires that `id` identifies a valid credentials cache and `*cursor` be a cursor returned by **krb5_cc_start_seq_get**() or a subsequent call to **krb5_cc_next_cred**().

Errors: may return error code if `*cursor` is invalid.

krb5_error_code cc_remove_cred
krb5_cc_remove_cred(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_ccache `id`,
krb5_flags `which`,
krb5_creds * `cred`)

Removes any credentials from `id` which match the principal name cred-¿server and the fields in `cred` masked by `which`. Requires that `id` identifies a valid credentials cache.

Errors: returns error code if nothing matches; returns error code if couldn't delete.

krb5_error_code cc_set_flags
krb5_cc_set_flags(*/* IN/OUT */*
krb5_context `context`,
krb5_ccache `id`,
/ IN */*
krb5_flags `flags`)

Sets the flags on the cache `id` to `flags`. Useful flags are defined in `<krb5.h>`.

unsigned int get_notification_message
krb5_get_notification_message()

Intended for use by Windows. Will register a unique message type using **RegisterWindowMessage**() which will be notified whenever the cache changes. This will allow all processes to recheck their caches.

4.3 Replay cache functions

The replay cache functions deal with verifying that AP_REQ's do not contain duplicate authenticators; the storage must be non-volatile for the site-determined validity period of authenticators.

Each replay cache has a string "name" associated with it. The use of this name is dependent on the underlying caching strategy (for file-based things, it would be a cache file name). The caching strategy uses non-volatile storage so that replay integrity can be maintained across system failures.

auth_to_rep

krb5_error_code
krb5_auth_to_rep(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_tkt_authent * `auth`,
/ OUT */*
krb5_donot_replay * `rep`)

Extract the relevant parts of `auth` and fill them into the structure pointed to by `rep`. `rep->client` and `rep->server` are set to allocated storage and should be freed when `*rep` is no longer needed.

rc_resolve_full

krb5_error_code
krb5_rc_resolve_full(*/* IN/OUT */*
krb5_context `context`,
krb5_rcache * `id`,
/ IN */*
char * `string_name`)

`id` is filled in to identify a replay cache which corresponds to the name in `string_name`. The cache is not opened. Requires that `string_name` be of the form "type:residual" and that "type" is a type known to the library.

Before the cache can be used **krb5_rc_initialize**() or **krb5_rc_recover**() must be called.

Errors: error if cannot resolve name.

rc_resolve_type

krb5_error_code
krb5_rc_resolve_type(*/* IN/OUT */*
krb5_context `context`,
krb5_rcache * `id`,
/ IN */*
char * `type`)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Looks up `type` in the list of knows cache types and if found attaches the operations to `*id` which must be previously allocated.

If `type` is not found, KRB5_RC_TYPE_NOTFOUND is returned.

rc_register_type

krb5_error_code
krb5_rc_register_type(*/* IN */*
krb5_context `context`,
krb5_rc_ops * `ops`)

Adds a new replay cache type implemented and identified by `ops` to the set recognized by **krb5_rc_resolve**(). This function requires that a ticket cache of the type named in `ops->prefix` has not been previously registered.

char * rc_default_name
krb5_rc_default_name(*/* IN */*
krb5_context `context`)

Returns the name of the default replay cache; this may be equivalent to **getenv**(`"KRB5RCACHE"`) with an appropriate fallback.

char * rc_default_type
krb5_rc_default_type(*/* IN */*
krb5_context `context`)

Returns the type of the default replay cache.

krb5_error_code rc_default
krb5_rc_default(*/* IN/OUT */*
krb5_context `context`,
krb5_rcache * `id`)

This function returns an unopened replay cache of the default type and default name (as would be returned by **krb5_rc_default_type**() and **krb5_rc_default_name**()). Before the cache can be used **krb5_rc_initialize**() or **krb5_rc_recover**() must be called.

krb5_error_code rc_initialize
krb5_rc_initialize(*/* IN */*
krb5_context `context`,
krb5_rcache `id`,
krb5_deltat `auth_lifespan`)

Creates/refreshes the replay cache identified by `id` and sets its authenticator lifespan to `auth_lifespan`. If the replay cache already exists, its contents are destroyed.

Errors: permission errors, system errors

krb5_error_code rc_recover
krb5_rc_recover(*/* IN */*
krb5_context `context`,
krb5_rcache `id`)

Attempts to recover the replay cache `id`, (presumably after a system crash or server restart).

Errors: error indicating that no cache was found to recover

krb5_error_code rc_destroy
krb5_rc_destroy(*/* IN */*
krb5_context `context`,
krb5_rcache `id`)

Destroys the replay cache id. Requires that id identifies a valid replay cache.

Errors: permission errors.

rc_close

krb5_error_code
krb5_rc_close(*/* IN */*
krb5_context context,
krb5_rcache id)

Closes the replay cache id, invalidates id, and releases any other resources acquired during use of the replay cache. Requires that id identifies a valid replay cache.

Errors: permission errors

rc_store

krb5_error_code
krb5_rc_store(*/* IN */*
krb5_context context,
krb5_rcache id,
krb5_donot_replay * rep)

Stores rep in the replay cache id. Requires that id identifies a valid replay cache.

Returns KRB5KRB_AP_ERR_REPEAT if rep is already in the cache. May also return permission errors, storage failure errors.

rc_expunge

krb5_error_code
krb5_rc_expunge(*/* IN */*
krb5_context context,
krb5_rcache id)

Removes all expired replay information (i.e. those entries which are older than then authenticator lifespan of the cache) from the cache id. Requires that id identifies a valid replay cache.

Errors: permission errors.

rc_get_lifespan

krb5_error_code
krb5_rc_get_lifespan(*/* IN */*
krb5_context context,
krb5_rcache id,
/ OUT */*
krb5_deltat * auth_lifespan)

Fills in auth_lifespan with the lifespan of the cache id. Requires that id identifies a valid replay cache.

rc_resolve

krb5_error_code
krb5_rc_resolve(*/* IN/OUT */*
krb5_context context,
krb5_rcache id,
/ IN */*
char * name)

Initializes private data attached to `id`. This function MUST be called before the other per-replay cache functions.

Requires that `id` points to allocated space, with an initialized `id->ops` field.

Since **krb5_rc_resolve**() allocates memory, **krb5_rc_close**() must be called to free the allocated memory, even if neither **krb5_rc_initialize**() or **krb5_rc_recover**() were successfully called by the application.

Returns: allocation errors.

char * rc_get_name
krb5_rc_get_name(*/* IN */*
krb5_context `context`,
krb5_rcache `id`)

Returns the name (excluding the type) of the rcache `id`. Requires that `id` identifies a valid replay cache.

char * rc_get_type
krb5_rc_get_type(*/* IN */*
krb5_context `context`,
krb5_rcache `id`)

Returns the type (excluding the name) of the rcache `id`. Requires that `id` identifies a valid replay cache.

4.4 Key table functions

The key table functions deal with storing and retrieving service keys for use by unattended services which participate in authentication exchanges.

Keytab routines are all be atomic. Every routine that acquires a non-sharable resource releases it before it returns.

All keytab types support multiple concurrent sequential scans.

The order of values returned from **krb5_kt_next_entry**() is unspecified.

Although the "right thing" should happen if the program aborts abnormally, a close routine, **krb5_kt_free_entry**(), is provided for freeing resources, etc. People should use the close routine when they are finished.

krb5_error_code kt_register
krb5_kt_register(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_kt_ops * `ops`)

Adds a new ticket cache type to the set recognized by **krb5_kt_resolve**(). Requires that a keytab type named `ops->prefix` is not yet known.

An error is returned if `ops->prefix` is already known.

kt_resolve

krb5_error_code
krb5_kt_resolve(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const char * `string_name`,
/ OUT */*
krb5_keytab * `id`)

Fills in `*id` with a handle identifying the keytab with name "string_name". The keytab is not opened. Requires that `string_name` be of the form "type:residual" and "type" is a type known to the library.

Errors: badly formatted name.

kt_default_name

krb5_error_code
krb5_kt_default_name(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
char * `name`,
int `namesize`)

`name` is filled in with the first `namesize` bytes of the name of the default keytab. If the name is shorter than `namesize`, then the remainder of `name` will be zeroed.

kt_default

krb5_error_code
krb5_kt_default(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_keytab * `id`)

Fills in `id` with a handle identifying the default keytab.

kt_read_service_key

krb5_error_code
krb5_kt_read_service_key(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_pointer `keyprocarg`,
krb5_principal `principal`,
krb5_kvno `vno`,
krb5_keytype `keytype`,
/ OUT */*
krb5_keyblock ** `key`)

If **keyprocarg**() is not NULL, it is taken to be a **char *** denoting the name of a keytab. Otherwise, the default keytab will be used. The keytab is opened and searched for the entry identified by `principal`, `keytype`, and `vno`, returning the resulting key in `*key` or returning an error code if it is not found.

krb5_free_keyblock() should be called on `*key` when the caller is finished with the key.

Returns an error code if the entry is not found.

krb5_error_code kt_add_entry
krb5_kt_add_entry(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_keytab `id`,
krb5_keytab_entry * `entry`)

Calls the keytab-specific add routine **krb5_kt_add_internal**() with the same function arguments. If this routine is not available, then KRB5_KT_NOWRITE is returned.

krb5_error_code kt_remove_entry
krb5_kt_remove_entry(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
krb5_keytab `id`,
krb5_keytab_entry * `entry`)

Calls the keytab-specific remove routine **krb5_kt_remove_internal**() with the same function arguments. If this routine is not available, then KRB5_KT_NOWRITE is returned.

krb5_error_code kt_get_name
krb5_kt_get_name(*/* IN/OUT */*
krb5_context `context`,
krb5_keytab `id`,
/ OUT */*
char * `name`,
/ IN */*
int `namesize`)

,
name is filled in with the first `namesize` bytes of the name of the keytab identified by **id**(). If the name is shorter than `namesize`, then ,
name will be null-terminated.

krb5_error_code kt_close
krb5_kt_close(*/* IN/OUT */*
krb5_context `context`,
krb5_keytab `id`)

Closes the keytab identified by `id` and invalidates `id`, and releases any other resources acquired during use of the key table.

Requires that `id` identifies a keytab.

kt_get_entry

```
krb5_error_code
krb5_kt_get_entry(/* IN/OUT */
                  krb5_context context,
                  krb5_keytab id,
                  /* IN */
                  krb5_principal principal,
                  krb5_kvno vno,
                  krb5_keytype keytype,
                  /* OUT */
                  krb5_keytab_entry * entry)
```

Searches the keytab identified by `id` for an entry whose principal matches `principal`, whose keytype matches `keytype`, and whose key version number matches `vno`. If `vno` is zero, the first entry whose principal matches is returned.

Returns an error code if no suitable entry is found. If an entry is found, the entry is returned in `*entry`; its contents should be deallocated by calling **krb5_kt_free_entry**() when no longer needed.

kt_free_entry

```
krb5_error_code
krb5_kt_free_entry(/* IN/OUT */
                   krb5_context context,
                   krb5_keytab_entry * entry)
```

Releases all storage allocated for `entry`, which must point to a structure previously filled in by **krb5_kt_get_entry**() or **krb5_kt_next_entry**().

kt_start_seq_get

```
krb5_error_code
krb5_kt_start_seq_get(/* IN/OUT */
                      krb5_context context,
                      krb5_keytab id,
                      /* OUT */
                      krb5_kt_cursor * cursor)
```

Prepares to read sequentially every key in the keytab identified by `id`. `cursor` is filled in with a cursor to be used in calls to **krb5_kt_next_entry**().

kt_next_entry

```
krb5_error_code
krb5_kt_next_entry(/* IN/OUT */
                   krb5_context context,
                   krb5_keytab id,
                   /* OUT */
                   krb5_keytab_entry * entry,
                   /* IN/OUT */
                   krb5_kt_cursor * cursor)
```

Fetches the "next" entry in the keytab, returning it in `*entry`, and updates `*cursor` for the next request. If the keytab changes during the sequential get, an error is guaranteed. `*entry` should be freed after use by calling **krb5_kt_free_entry**().

Requires that `id` identifies a valid keytab. and `*cursor` be a cursor returned by **krb5_kt_start_seq_get**() or a subsequent call to **krb5_kt_next_entry**().

Errors: error code if no more cache entries or if the keytab changes.

krb5_error_code
krb5_kt_end_seq_get(*/* IN/OUT */*
krb5_context `context`,
krb5_keytab `id`,
krb5_kt_cursor * `cursor`) kt_end_seq_get

Finishes sequential processing mode and invalidates `cursor`, which must never be re-used after this call.

Requires that `id` identifies a valid keytab and `*cursor` be a cursor returned by **krb5_kt_start_seq_get**() or a subsequent call to **krb5_kt_next_entry**().

May return error code if `cursor` is invalid.

4.5 Free functions

The free functions deal with deallocation of memory that has been allocated by various routines. It is recommended that the developer use these routines as they will know about the contents of the structures.

void
krb5_xfree(*/* IN/OUT */*
void * `ptr`) xfree

Frees the pointer ,
ptr . This is a wrapper macro to **free**() that is designed to keep lint "happy."

void
krb5_free_data(*/* IN/OUT */*
krb5_context `context`,
krb5_data * `val`) free_data

Frees the data structure `val`, including the pointer `val` which has been allocate by any of numerous routines.

void
krb5_free_princial(*/* IN/OUT */*
krb5_context `context`,
krb5_principal `val`) free_princial

Frees the pwd_data `val` that has been allocated from **krb5_copy_principal**().

void
krb5_free_authenticator(*/* IN/OUT */*
krb5_context `context`,
krb5_authenticator * `val`) free_authenticator

Frees the authenticator `val`, including the pointer `val`.

free_authenticator_contents

void
krb5_free_authenticator_contents(*/* IN/OUT */*
krb5_context `context`,
krb5_authenticator * `val`)

Frees the authenticator contents of `val`. The pointer `val` is not freed.

free_addresses

void
krb5_free_addresses(*/* IN/OUT */*
krb5_context `context`,
krb5_address ** `val`)

Frees the series of addresses `*val` that have been allocated from **krb5_copy_addresses**().

free_address

void
krb5_free_address(*/* IN/OUT */*
krb5_context `context`,
krb5_address * `val`)

Frees the address `val`.

free_authdata

void
krb5_free_authdata(*/* IN/OUT */*
krb5_context `context`,
krb5_authdata ** `val`)

Frees the authdata structure pointed to by `val` that has been allocated from **krb5_copy_authdata**().

free_enc_tkt_part

void
krb5_free_enc_tkt_part(*/* IN/OUT */*
krb5_context `context`,
krb5_enc_tkt_part * `val`)

Frees `val` that has been allocated from **krb5_enc_tkt_part**() and **krb5_decrypt_tkt_part**().

free_ticket

void
krb5_free_ticket(*/* IN/OUT */*
krb5_context `context`,
krb5_ticket * `val`)

Frees the ticket `val` that has been allocated from **krb5_copy_ticket**() and other routines.

free_tickets

void
krb5_free_tickets(*/* IN/OUT */*
krb5_context `context`,
krb5_ticket ** `val`)

Frees the tickets pointed to by `val`.

void free_kdc_req
krb5_free_kdc_req(*/* IN/OUT */*
krb5_context `context`,
krb5_kdc_req * `val`)

Frees the kdc_req `val` and all substructures. The pointer `val` is freed as well.

void free_kdc_rep
krb5_free_kdc_rep(*/* IN/OUT */*
krb5_context `context`,
krb5_kdc_rep * `val`)

Frees the kdc_rep `val` that has been allocated from **krb5_get_in_tkt**().

void free_kdc_rep_part
krb5_free_kdc_rep_part(*/* IN/OUT */*
krb5_context `context`,
krb5_enc_kdc_rep_part * `val`)

Frees the kdc_rep_part `val`.

void free_error
krb5_free_error(*/* IN/OUT */*
krb5_context `context`,
krb5_error * `val`)

Frees the error `val` that has been allocated from **krb5_read_error**() or **krb5_sendauth**().

void free_ap_req
krb5_free_ap_req(*/* IN/OUT */*
krb5_context `context`,
krb5_ap_req * `val`)

Frees the ap_req `val`.

void free_ap_rep
krb5_free_ap_rep(*/* IN/OUT */*
krb5_context `context`,
krb5_ap_rep * `val`)

Frees the ap_rep `val`.

void free_safe
krb5_free_safe(*/* IN/OUT */*
krb5_context `context`,
krb5_safe * `val`)

Frees the safe application data `val` that is allocated with `decode_krb5_safe`.

free_priv

```
void
krb5_free_priv(/* IN/OUT */
               krb5_context context,
               krb5_priv * val)
```

Frees the private data `val` that has been allocated from **decode_krb5_priv**().

free_priv_enc_part

```
void
krb5_free_priv_enc_part(/* IN/OUT */
                        krb5_context context,
                        krb5_priv_enc_part * val)
```

Frees the private encoded part `val` that has been allocated from **decode_krb5_enc_priv_part**().

free_cred

```
void
krb5_free_cred(/* IN/OUT */
               krb5_context context,
               krb5_cred * val)
```

Frees the credential `val`.

free_creds

```
void
krb5_free_creds(/* IN/OUT */
                krb5_context context,
                krb5_creds * val)
```

Calls **krb5_free_cred_contents**() with `val` as the argument. `val` is freed as well.

free_cred_contents

```
void
krb5_free_cred_contents(/* IN/OUT */
                        krb5_context context,
                        krb5_creds * val)
```

The function zeros out the session key stored in the credential and then frees the credentials structures. The argument `val` is **not** freed.

free_cred_enc_part

```
void
krb5_free_cred_enc_part(/* IN/OUT */
                        krb5_context context,
                        krb5_cred_enc_part * val)
```

Frees the addresses and ticket_info elements of `val`. `val` is **not** freed by this routine.

free_checksum

```
void
krb5_free_checksum(/* IN/OUT */
                   krb5_context context,
                   krb5_checksum * val)
```

The checksum and the pointer `val` are both freed.

void free_keyblock
krb5_free_keyblock(*/* IN/OUT */*
krb5_context `context`,
krb5_keyblock * `val`)

The keyblock contents of `val` are zeroed and the memory freed. The pointer `val` is freed as well.

void free_pa_data
krb5_free_pa_data(*/* IN/OUT */*
krb5_context `context`,
krb5_pa_data ** `val`)

Frees the contents of `*val`. `val` is freed as well.

void free_ap_rep_enc_part
krb5_free_ap_rep_enc_part(*/* IN/OUT */*
krb5_context `context`,
krb5_ap_rep_enc_part * `val`)

Frees the subkey keyblock (if set) as well as `val` that has been allocated from **krb5_rd_rep**() or **krb5_send_auth**().

void free_tkt_authent
krb5_free_tkt_authent(*/* IN/OUT */*
krb5_context `context`,
krb5_tkt_authent * `val`)

Frees the ticket and authenticator portions of `val`. The pointer `val` is freed as well.

void free_pwd_data
krb5_free_pwd_data(*/* IN/OUT */*
krb5_context `context`,
passwd_pwd_data * `val`)

Frees the pwd_data `val` that has been allocated from **decode_krb5_pwd_data**().

void free_pwd_sequences
krb5_free_pwd_sequences(*/* IN/OUT */*
krb5_context `context`,
passwd_phrase_element ** `val`)

Frees the passwd_phrase_element `val`. This is usually called from **krb5_free_pwd_data**().

void free_realm_tree
krb5_free_realm_tree(*/* IN/OUT */*
krb5_context `context`,
krb5_principal * `realms`)

Frees the realms tree `realms` returned by **krb5_walk_realm_tree**().

free_tgt_creds

void
krb5_free_tgt_creds(*/* IN/OUT */*
krb5_context `context`,
krb5_creds ** `tgts`)

Frees the TGT credentials `tgts` returned by **krb5_get_cred_from_kdc**().

4.6 Operating-system specific functions

The operating-system specific functions provide an interface between the other parts of the `libkrb5.a` libraries and the operating system.

Beware! Any of the functions below are allowed to be implemented as macros. Prototypes for functions can be found in `<krb5.h>`; other definitions (including macros, if used) are in `<krb5/libos.h>`.

The following global symbols are provided in `libos.a`. If you wish to substitute for any of them, you must substitute for all of them (they are all declared and initialized in the same object file):

extern char *`krb5_defkeyname`**:** default name of key table file

extern char *`krb5_lname_file`**:** name of aname/lname translation database

extern int `krb5_max_dgram_size`**:** maximum allowable datagram size

extern int `krb5_max_skdc_timeout`**:** maximum per-message KDC reply timeout

extern int `krb5_skdc_timeout_shift`**:** shift factor (bits) to exponentially back-off the KDC timeouts

extern int `krb5_skdc_timeout_1`**:** initial KDC timeout

extern char *`krb5_kdc_udp_portname`**:** name of KDC UDP port

extern char *`krb5_default_pwd_prompt1`**:** first prompt for password reading.

extern char *`krb5_default_pwd_prompt2`**:** second prompt

4.6.1 Operating specific context

The **krb5_context** has space for operating system specific data. These functions are called from **krb5_init_context**() and **krb5_free_context**(), but are included here for completeness.

os_init_context

krb5_error_code
krb5_os_init_context(*/* IN/OUT */*
krb5_context `context`)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Initializes `context->os_context` and establishes the location of the initial configuration files.

krb5_error_code os_free_context
krb5_os_free_context(*/* IN/OUT */*
krb5_context `context`)

NOTE: This is an internal function, which is not necessarily intended for use by application programs. Its interface may change at any time.

Frees the operating system specific portion of `context`.

4.6.2 Configuration based functions

These functions allow access to configuration specific information. In some cases, the configuration may be overriden by program control.

krb5_error_code set_config_files
krb5_set_config_files(*/* IN/OUT */*
krb5_context `context`,
/ IN */*
const char ** `filenames`)

Sets the list of configuration files to be examined in determining machine defaults. `filenames` is an array of files to check in order. The array must have a NULL entry as the last element.

Returns system errors.

krb5_error_code get_krbhst
krb5_get_krbhst(*/* IN */*
krb5_context `context`,
const krb5_data * `realm`,
/ OUT */*
char *** `hostlist`)

Figures out the Kerberos server names for the given `realm`, filling in `hostlist` with a null terminated array of pointers to hostnames.

If `realm` is unknown, the filled-in pointer is set to NULL.

The pointer array and strings pointed to are all in allocated storage, and should be freed by the caller when finished.

Returns system errors.

krb5_error_code free_krbhst
krb5_free_krbhst(*/* IN */*
krb5_context `context`,
char * const * `hostlist`)

Frees the storage taken by a host list returned by **krb5_get_krbhst**().

get_default_realm

krb5_error_code
krb5_get_default_realm(/* *IN* */
krb5_context `context`,
/* *OUT* */
char ** `lrealm`)

Retrieves the default realm to be used if no user-specified realm is available (e.g. to interpret a user-typed principal name with the realm omitted for convenience), filling in `lrealm` with a pointer to the default realm in allocated storage.

It is the caller's responsibility for freeing the allocated storage pointed to be `lream` when it is finished with it.

Returns system errors.

set_default_realm

krb5_error_code
krb5_set_default_realm(/* *IN* */
krb5_context `context`,
char * `realm`)

Sets the default realm to be used if no user-specified realm is available (e.g. to interpret a user-typed principal name with the realm omitted for convenience). (c.f. krb5_get_default_realm)

If `realm` is NULL, then the operating system default value will used.

Returns system errors.

get_host_realm

krb5_error_code
krb5_get_host_realm(/* *IN* */
krb5_context `context`,
const char * `host`,
/* *OUT* */
char *** `realmlist`)

Figures out the Kerberos realm names for `host`, filling in `realmlist` with a pointer to an argv[] style list of names, terminated with a null pointer.

If `host` is NULL, the local host's realms are determined.

If there are no known realms for the host, the filled-in pointer is set to NULL.

The pointer array and strings pointed to are all in allocated storage, and should be freed by the caller when finished.

Returns system errors.

free_host_realm

krb5_error_code
krb5_free_host_realm(/* *IN* */
krb5_context `context`,
char * const * `realmlist`)

Frees the storage taken by a `realmlist` returned by **krb5_get_local_realm**().

krb5_error_code get_realm_domain
krb5_get_realm_domain(*/* IN/OUT */*
krb5_context context,
/ IN */*
const char * realm,
/ OUT */*
char ** domain)

Determines the proper name of a realm. This is mainly so that a krb4 principal can be converted properly into a krb5 one. If realm is null, the function will assume the default realm of the host. The returned *domain is allocated and must be freed by the caller.

4.6.3 Disk based functions

These functions all relate to disk based I/O.

krb5_error_code lock_file
krb5_lock_file(*/* IN */*
krb5_context context,
in fd,
int mode)

Attempts to lock the file in the given mode; returns 0 for a successful lock, or an error code otherwise.

The caller should arrange for the file referred by fd to be opened in such a way as to allow the required lock.

Modes are given in <krb5/libos.h>

krb5_error_code unlock_file
krb5_unlock_file(*/* IN */*
krb5_context context,
int fd)

Attempts to (completely) unlock the file. Returns 0 if successful, or an error code otherwise.

krb5_error_code create_secure_file
krb5_create_secure_file(*/* IN */*
krb5_context context,
const char * pathname)

Creates a file named pathname which can only be read by the current user.

krb5_error_code sync_disk_file
krb5_sync_disk_file(*/* IN */*
krb5_context context,
FILE * fp)

Assures that the changes made to the file pointed to by the file handle fp are forced out to disk.

4.6.4 Network based routines

These routines send and receive network data the specifics of addresses and families on a given operating system.

os_localaddr

krb5_error_code
krb5_os_localaddr(*/* IN */*
krb5_context `context`,
/ OUT */*
krb5_address *** `addr`)

Return all the protocol addresses of this host.

Compile-time configuration flags will indicate which protocol family addresses might be returned. `*addr` is filled in to point to an array of address pointers, terminated by a null pointer. All the storage pointed to is allocated and should be freed by the caller with **krb5_free_address**() when no longer needed.

gen_portaddr

krb5_error_code
krb5_gen_portaddr(*/* IN */*
krb5_context `context`,
const krb5_address * `adr`,
krb5_const_pointer `ptr`,
/ OUT */*
krb5_address ** `outaddr`)

Given an address `adr` and an additional address-type specific portion pointed to by `port` this routine combines them into a freshly-allocated **krb5_address** with type **ADDRTYPE_ADDRPORT** and fills in `*outaddr` to point to this address. For IP addresses, `ptr` should point to a network-byte-order TCP or UDP port number. Upon success, `*outaddr` will point to an allocated address which should be freed with **krb5_free_address**().

sendto_kdc

krb5_error_code
krb5_sendto_kdc(*/* IN */*
krb5_context `context`,
const krb5_data * `send`,
const krb5_data * `realm`,
/ OUT */*
krb5_data * `receive`)

Send the message `send` to a KDC for realm `realm` and return the response (if any) in `receive`.

If the message is sent and a response is received, 0 is returned, otherwise an error code is returned.

The storage for `receive` is allocated and should be freed by the caller when finished.

int net_read
krb5_net_read(*/* IN */*
krb5_context context,
int fd,
/ OUT */*
char * buf,
/ IN */*
int len)

Like read(2), but guarantees that it reads as much as was requested or returns -1 and sets errno.

(make sure your sender will send all the stuff you are looking for!) Only useful on stream sockets and pipes.

int net_write
krb5_net_write(*/* IN */*
krb5_context context,
int fd,
const char * buf,
int len)

Like write(2), but guarantees that it writes as much as was requested or returns -1 and sets errno.

Only useful on stream sockets and pipes.

krb5_error_code write_message
krb5_write_message(*/* IN */*
krb5_context context,
krb5_pointer fd,
krb5_data * data)

krb5_write_message() writes data to the network as a message, using the network connection pointed to by `fd`.

krb5_error_code read_message
krb5_read_message(*/* IN */*
krb5_context context,
krb5_pointer fd,
/ OUT */*
krb5_data * data)

Reads data from the network as a message, using the network connection pointed to by fd.

4.6.5 Operating specific access functions

These functions are involved with access control decisions and policies.

aname_to_localname

krb5_error_code
krb5_aname_to_localname(*/* IN */*
krb5_context context,
krb5_const_principal aname,
int lnsize,
/ OUT */*
char * lname)

Converts a principal name aname to a local name suitable for use by programs wishing a translation to an environment-specific name (e.g. user account name).

lnsize specifies the maximum length name that is to be filled into lname. The translation will be null terminated in all non-error returns.

Returns system errors.

kuserok

krb5_boolean
krb5_kuserok(*/* IN */*
krb5_context context,
krb5_principal principal,
const char * luser)

Given a Kerberos principal principal, and a local username luser, determine whether user is authorized to login to the account luser. Returns TRUE if authorized, FALSE if not authorized.

sname_to_principal

krb5_error_code
krb5_sname_to_principal(*/* IN */*
krb5_context context,
const char * hostname,
const char * sname,
krb5_int32 type,
/ OUT */*
krb5_principal * ret_princ)

Given a hostname hostname and a generic service name sname, this function generates a full principal name to be used when authenticating with the named service on the host. The full prinicpal name is returned in ret_princ.

The realm of the principal is determined internally by calling **krb5_get_host_realm**().

The type argument controls how **krb5_sname_to_principal**() generates the principal name, ret_princ, for the named service, sname. Currently, two values are supported: KRB5_NT_SRV_HOST, and KRB5_NT_UNKNOWN.

If type is set to KRB5_NT_SRV_HOST, the hostname will be canonicalized, i.e. a fully qualified lowercase hostname using the primary name and the domain name, before ret_princ is generated in the form "sname/hostname@LOCAL.REALM." Most applications should use KRB5_NT_SRV_HOST.

However, if type is set to KRB5_NT_UNKNOWN, while the generated principal name will have the form "sname/hostname@LOCAL.REALM" the hostname will not be canonicalized first. It will appear exactly as it was passed in hostname.

The caller should release ret_princ's storage by calling **krb5_free_principal**()

when it is finished with the principal.

4.6.6 Miscellaneous operating specific functions

These functions handle the other operating specific functions that do not fall into any other major class.

krb5_error_code timeofday
krb5_timeofday(*/* IN */*
krb5_context context,
/ OUT */*
krb5_context context,
krb5_int32 * timeret)

Retrieves the system time of day, in seconds since the local system's epoch. [The ASN.1 encoding routines must convert this to the standard ASN.1 encoding as needed]

krb5_error_code us_timeofday
krb5_us_timeofday(*/* IN */*
krb5_context context,
/ OUT */*
krb5_int32 * seconds,
krb5_int32 * microseconds)

Retrieves the system time of day, in seconds since the local system's epoch. [The ASN.1 encoding routines must convert this to the standard ASN.1 encoding as needed]

The seconds portion is returned in `*seconds`, the microseconds portion in `*microseconds`.

krb5_error_code read_password
krb5_read_password(*/* IN */*
krb5_context context,
const char * prompt,
const char * prompt2,
/ OUT */*
char * return_pwd,
/ IN/OUT */*
int * size_return)

Read a password from the keyboard. The first `*size_return` bytes of the password entered are returned in `return_pwd`. If fewer than `*size_return` bytes are typed as a password, the remainder of `return_pwd` is zeroed. Upon success, the total number of bytes filled in is stored in `*size_return`.

`prompt` is used as the prompt for the first reading of a password. It is printed to the terminal, and then a password is read from the keyboard. No newline or spaces are emitted between the prompt and the cursor, unless the newline/space is included in the prompt.

If `prompt2` is a null pointer, then the password is read once. If `prompt2` is set, then it is used as a prompt to read another password in the same manner as described for `prompt`. After the second password is read, the two passwords are compared, and an error is returned if they are not identical.

Echoing is turned off when the password is read.

If there is an error in reading or verifying the password, an error code is returned; else zero is returned.

random_confounder

krb5_error_code
krb5_random_confounder(/* *IN* */
krb5_context `context`,
int `size`,
/* *OUT* */
krb5_pointer `fillin`)

Given a length and a pointer, fills in the area pointed to by `fillin` with `size` random octets suitable for use in a confounder.

gen_replay_name

krb5_error_code
krb5_gen_replay_name(/* *IN* */
krb5_context `context`,
const krb5_address * `inaddr`,
const char * `uniq`,
/* *OUT* */
char ** `string`)

Given a **krb5_address** with type **ADDRTYPE_ADDRPORT** in `inaddr`, this function unpacks its component address and additional type, and uses them along with `uniq` to allocate a fresh string to represent the address and additional information. The string is suitable for use as a replay cache tag. This string is allocated and should be freed with **free**() when the caller has finished using it. When using IP addresses, the components in `inaddr->contents` must be of type **ADDRTYPE_INET** and **ADDRTYPE_PORT**.

Index