# Using Kerberos from Python

*Release 0.1*

Fred L. Drake, Jr. and Roger E. Masse

May 15, 1998

Corporation for National Research Initiatives
1895 Preston White Dr., Reston, Va 20191
E-mail: fdrake@cnri.reston.va.us, rmasse@cnri.reston.va.us



**Abstract**

This document describes modules developed to allow access to Kerberos V5 from Python. We have developed a object-oriented approach to using some Kerberos V5 facilities which allows much of the tedium to be hidden from the application programmer. A wrapper module for Python's `socket` module provides an integration which is largely transparent.

## Contents

# 1 Extension Module krb5

The krb5 module provides Kerberos V5 services to Python. The detailed Kerberos API is hidden behind a more intelligible API that can be used from Python. Understanding Kerberos is required before this module can be used effectively.

Only two objects really need to be exposed to the Python programmer: the Principal and the Authentication Context. The Authentication Context comes in two varieties, one for use with UDP-based channels and another for TCP connections. The API for the TCP variety is a superset of the API for the UDP flavor.

For all krb5 operations, a single exception has been defined:

**Krb5Error**
: Raised when a Kerberos-specific error is raised. Instances have two attributes: err_code is the numeric error code returned by the Kerberos library, and message is the standard error message provided by the error_message() function used within Kerberos. These values may also be retrieved by unpacking the exception instance:

```
try:
    krb5.send_auth(...)
except krb5.Krb5Error, (err_code, message):
    print "Failed:", message
```

Some Krb5Error exceptions are raised with an err_code of 0; these are not a mistake! These are real errors which don't have specific error code assignments from Kerberos. The message attribute holds an explanation in all cases.

Note that socket.error may be raised by some functions and methods if appropriate.

## 1.1 Kerberos Principals

Three functions are provided to support retrieval of Kerberos Principal objects. All of these return objects of the type PrincipalType.

**get_principal**(*name*)
: Return a Principal object corresponding to a fully-formed string representation, such as 'splat@REALM.WE.LIVE.IN'. If the name is not parsable, Krb5Error is raised with an err_code attribute of KRB5_PARSE_MALFORMED.

**get_login_principal**()
: Retrieve the default principal.

**get_svc_principal**(*svcname*[, *hostname*])
: Return a principal for a specific service. If *hostname* is omitted or the empty string, the local hostname is used. Note that Kerberos does not in any way validate that the specified service exists.

**PrincipalType**
: The type of the Principal objects returned from get_principal(), get_login_principal(), and get_svc_principal().

Principal objects provide some useful methods and data attributes:

**name**()
: Return the fully-formed name of the principal, suitable for passing to get_principal() at a later time. The name includes the realm.

**realm**()
: Return only the realm portion of the Principal name.

**host**

For principals created using `get_svc_principal()`, this is the fully-qualified domain name for the machine on which the service should be found. For all other principals, this is `None`.

**`service`**
For principals created using `get_svc_principal()`, this is the name of the serivce passed in. For all other principals, this is `None`.

## 1.2 Authentication Contexts

### Common Methods and Attributes

**`getflags`**`()`
Return the bitwise-or of the current operational parameters for the Authentication Context. See the `set-flags()` method.

**`setflags`**`(`*flags*`)`
Set operational parameters for the the Authentication Context. Valid flags supported for this module include `KRB5_AUTH_CONTEXT_DO_TIME` and `KRB5_AUTH_CONTEXT_DO_SEQUENCE`. Parameters to use should be bitwise-`or`'ed together and passed as *flags*.

**`mk_safe`**`(`*string*`)`
Return an encoded version of string which includes an encrypted checksum of *string*; only the remote principal should be able to decode the checksum and be able to verify that *string* actually arrived from the sender unmodified. This does not provide privacy.

**`rd_safe`**`(`*data*`)`
Return a string encapsulated in *data* with an encrypted checksum and verify the checksum and sender identity. If the checksum or sender doesn't match, `Krb5Error` is raised.

**`mk_priv`**`(`*string*`)`
Return an encoded version of string which includes an encrypted form of *string*; only the remote principal should be able to decode the original string and be able to verify that *string* actually arrived from the sender unmodified.

**`rd_priv`**`(`*data*`)`
Return the plaintext encapsulated in *data* and verify the sender's identity. If the sender doesn't match, `Krb5Error` is raised.

These attributes of Authentication Contexts are provided on a read-only basis:

**`local`**
The local principal used for the authentication context.

**`remote`**
The remote principal used for the authentication context.

### UDPAuthenticationContext Constructors

These functions are used to construct Authentication Contexts for use with UDP sockets. Use the constructors described below for in association with TCP sockets.

**`mk_req`**`(`*sprinc, local_addr, remote_addr*`[`*, options*`]`)
Create an Authentication Context for use in communication with a remote server. This is used by a client initiating communication. The principal representing the server being contacted is passed as *sprinc*. The two addresses, *local_addr* and *remote_addr*, are both internet addresses represented as the tuple '`(`*host*`,` *port*`)`'. The application request options should be passed as a bitwise-`or` of option values in *options*. Valid options are `AP_OPTS_USE_SESSION_KEY` and `AP_OPTS_MUTUAL_REQUIRED`. The return value is a tuple of the Authentication Context and s string representing the first packet which should be sent to the server.

**rd_req**(*sprinc, data, local_addr, remote_addr*)

Create an Authentication Context based on data read from an incoming packet. This is used on a server to authenticate incoming communications. The server principal is passed as *sprinc* and the data from the first packet received from the client is passed as a string in *data*. The two address are as described for `mk_req()`.

**UDPAuthenticationContextType**

Type of the Authentication Context object returned by `mk_req()` and `rd_req()`.

### TCPAuthenticationContext Constructors

**send_auth**(*sock, cprinc, sprinc*[, *options*])

Begin a Kerberos authorization sequence (typically performed on the client). For the operation to complete, the corresponding server (on the other end of *sock*, a connected socket object) must do a corresponding `recv_auth()`. *sock* may be a Python socket object or an int object corresponding to *sock*`.fileno()`. *cprinc* and *sprinc* are the client and server Principal objects respectively. Subsequent calls to block read/write methods implicitly use *sock* for transport. The authentication context object is returned upon successful validation of the client and server principals

**recv_auth**(*sock, sprinc*)

Complete a Kerberos authorization sequence (typically performed on the server). `recv_auth()` blocks until a corresponding `send_auth()` is issued by the peer on the client end of the connected *sock*. *sprinc* is the server principal (for yourself). The authentication context object is returned upon successful validation of the client (read from *sock*) and server principals.

**TCPAuthenticationContextType**

Type of the Authentication Context object returned by `send_auth()` and `recv_auth()`.

### TCPAuthenticationContext Methods

The block reading and writing methods are unique to the TCP variant of the Authentication Context object because of the requirement that they operate only over a connected socket. These methods allow the sender to send arbitrarily large amounts of data with a single block_write and be assured that the receiver will allocate enough storage behind the scenes to complete the corresponding block read. These methods may raise `socket.error` on socket failures.

**block_read**()

Read the what was sent by a corresponding `block_write()`, `block_write_safe()`, or `block_write_priv()` call. the return value is a tuple containing the *type*, of the corresponding write from the set: `''`, `'safe'`, `'priv'`, followed by the read *data* as a string object.

**block_write**(*string*)

Write the contents of the argument over the connected socket. corresponding `block_read()` will return the type followed by the read string in the form of a tuple. If *string* equals `'Hello World'`, than the corresponding `block_read()` will return `('', 'Hello World')`. With *type* returned as the empty string, the indication is that no further Kerberos layer checking was performed apart from what was done during the call to the constructor `send_auth()` (initial handshake). This is the least secure and highest performing variant of the block write methods.

**block_write_safe**(*string*)

Write the contents of the argument over the connected socket. If *string* equals `'Hello World'`, then the corresponding `block_read()` will return the tuple: `('safe', 'Hello World')`. With *type* returned as `'safe'`, the indication is that the block was checksummed and the checksum data encrypted for transport and that the decrypted checksum matched that of the data read. (i.e the block was not tampered with) The data itself, however, was passed in the clear.

**block_write_priv**(*string*)

Write the contents of the argument over the connected socket. If *string* equals `'Hello World'`, then the

corresponding `block_read()` will return the tuple: `('priv', 'Hello World')`. With *type* returned as `'priv'`, the indication is that the block was encrypted for transport and successfully decrypted on by the reader. This is the most secure and lowest performance variant of the block write methods.

## 1.3 Utility Functions

Some utility functions provided by the Kerberos library are exported to the Python programmer. These functions allow the programmer to query certain aspects of the Kerberos configuration.

**`get_default_realm`**`()`
Returns the realm used when no realm is specified in a principal name.

**`get_krbhst`**`(`*realm*`)`
Returns a list of host names which will be used to locate the Key Distribution Center.

The following functions may raise `socket.error`. They are not particularly tied to Kerberos other than being provided by the Kerberos library.

**`net_read`**`(`*sock, length*`)`
Low level read function on a connected socket. `net_read()` is not an attribute of TCPAthenticationContext objects because it involves no Kerberos interaction. *sock* is a socket object or an integer corresponding to *sock*`.fileno()`. *length* is the amount that should be read and returned as a string (in bytes).

**`net_write`**`(`*sock, string*`)`
Low level write function to correspond with `net_write()`. *sock* is a socket object or an integer corresponding to *sock*`.fileno()`. *string* is the data to be written. `net_write()` is not an attribute of TCPAthentication-Context objects because it involves no Kerberos interaction.

## 1.4 Constants

Several constants are defined in the `krb5` module. Some are used for setting up the Authentication Context objects and other are used to test for specific values of `Krb5Error.err_code`.

### Application Request Options

These constants can be used with the *options* parameter to the constructors for the Authentication Context objects as described above.

**`AP_OPTS_USE_SESSION_KEY`**
Use a session key.

**`AP_OPTS_MUTUAL_REQUIRED`**
Require mutual authentication. Without this, the server is *not* required to authenticate itself with the client.

### Authentication Context Options

These options may be used to set and test flags on the Authentication Context objects using the `setflags()` and `getflags()` methods.

**`KRB5_AUTH_CONTEXT_DO_TIME`**
Use time stamps.

**`KRB5_AUTH_CONTEXT_DO_SEQUENCE`**
Use sequence numbers.

**KRB5_PROG_SUMTYPE_NOSUPP**
Requested checksum function is not supported.

**KRB5_SENDAUTH_REJECTED**
Server reject client connection.

**KRB5_FCC_NOFILE**
No credentials cache is available.

**KRB5_PARSE_MALFORMED**
Principal name could not be parsed by `get_principal()`.

**KRB5KRB_AP_ERR_BAD_INTEGRITY**
Integrity failure.

**KRB5KRB_AP_ERR_TKT_EXPIRED**
Ticket in credentials cache has expired; run **kinit** to update your tickets.

**KRB5KRB_AP_ERR_BADADDR**
Wrong address used to encode encrypted data.

**KRB5KRB_AP_ERR_BADORDER**
Wrong sequence number used to encode encrypted data.

# 2 Module `ksocket`

The `ksocket` module provides a socket-like object which supports alternate interfaces to provide Kerberos authentication for connections. This should be used when only authentication is used; most code which operates on a connected socket can use an instance of the `KStreamSocket` class implemented by this module. Only TCP sockets are currently supported.

A single exception is defined by this module:

**KSocketError**
This exception is raised for errors specifically involving the Kerberos aspect of the connection. Ordinary socket errors raise `socket.error`.

"Kerberized" sockets are created using this constructor:

**KStreamSocket**([*lprinc*])
Create a Kerberized socket. If *lprinc* is given, it is used as the local principal. If omitted, the result of `krb5.get_login_principal()` is used. The return value is an instance which behaves almost exactly like a socket, but with some additional interfaces and changes as described below.

## 2.1 KStreamSocket Methods

Most methods of `KStreamSocket` objects are identical to the socket methods of the same names; these are not described here. Refer to the `socket` module documentation for information on standard socket behavior.

The methods described here are either new with this class or have modified behavior.

**accept**()
Like the normal socket `accept()` method, but returns a Kerberized socket instead. The connect has already been authenticated once this method returns. This method does block while performing the Kerberos authentication handshake.

**bind**(*address*)

The socket address *address* must be a `(`*host*`, `*port*`)` pair. If *host* is the empty string (`''`), it will be replaced with the result of `socket.gethostname()` to ensure that the `getsockname()` method returns the same address as the `getpeername()` method of the remote connection. This is needed to make Kerberos work properly without making too many assumptions about system configuration.

**`connect`**`(`*address*`)`
Raises `KSocketError`; use `kconnect()` instead.

**`get_remote_principal`**`()`
Return the Kerberos principal for the remote side of the connection. If the socket is not connected, `KSocketError` is raised.

**`kconnect`**`(`*sprinc*, *port*[, *options*]`)`
Connect to a remote server. The server principal, given by *sprinc*, must be a service principal as returned by `krb5.get_svc_principal()`. If `type(`*sprinc*`)` is not `krb5.PrincipalType`, a `TypeError` exception is raised. If it does not have the required service information, `ValueError` is raised. Application request options can be passed in *options* if desired. Once the connection is established, the Kerberos authenticaion handshake is performed; this operation can block.