NAME

krb5_introduction - Introduction to the Kerberos 5 API

Kerberos 5 API Overview

All functions are documented in manual pages. This section tries to give an overview of the major components used in Kerberos library, and point to where to look for a specific function.

Kerberos context

A kerberos context (krb5_context) holds all per thread state. All global variables that are context specific are stored in this structure, including default encryption types, credential cache (for example, a ticket file), and default realms.

The internals of the structure should never be accessed directly, functions exist for extracting information.

See the manual page for **krb5_init_context()** how to create a context and module **Heimdal Kerberos 5 library** for more information about the functions.

Kerberos authentication context

Kerberos authentication context (krb5_auth_context) holds all context related to an authenticated connection, in a similar way to the kerberos context that holds the context for the thread or process.

The krb5_auth_context is used by various functions that are directly related to authentication between the server/client. Example of data that this structure contains are various flags, addresses of client and server, port numbers, keyblocks (and subkeys), sequence numbers, replay cache, and checksum types.

Kerberos principal

The Kerberos principal is the structure that identifies a user or service in Kerberos. The structure that holds the principal is the krb5_principal. There are function to extract the realm and elements of the principal, but most applications have no reason to inspect the content of the structure.

The are several ways to create a principal (with different degree of portability), and one way to free it.

See also the page **The principal handing functions.** for more information and also module **Heimdal Kerberos 5 principal functions**.

Credential cache

A credential cache holds the tickets for a user. A given user can have several credential caches, one for each realm where the user have the initial tickets (the first krbtgt).

The credential cache data can be stored internally in different way, each of them for different proposes. File credential (FILE) caches and processes based (KCM) caches are for permanent storage. While memory caches (MEMORY) are local caches to the local process.

Caches are opened with **krb5_cc_resolve()** or created with **krb5_cc_new_unique()**.

If the cache needs to be opened again (using **krb5_cc_resolve()**) **krb5_cc_close()** will close the handle, but not the remove the cache. **krb5_cc_destroy()** will zero out the cache, remove the cache so it can no longer be referenced.

See also **The credential cache functions** and **Heimdal Kerberos 5 credential cache functions** .

Kerberos errors

Kerberos errors are based on the com_err library. All error codes are 32-bit signed numbers, the first 24 bits define what subsystem the error originates from, and last 8 bits are 255 error codes within the library. Each error code have fixed string associated with it. For example, the error-code -1765328383 have the symbolic name KRB5KDC_ERR_NAME_EXP, and associated error string ‘‘Client’s entry in database has expired’’.

This is a great improvement compared to just getting one of the unix error-codes back. However, Heimdal have an extention to pass back customised errors messages. Instead of getting ‘‘Key table entry not found’’, the user might back ‘‘failed to find host/host.example.com@EXAMLE.COM(kvno 3) in keytab /etc/krb5.keytab (des-cbc-crc)’’. This improves the chance that the user find the cause of the error so you should use the customised error message whenever it’s available.

See also module **Heimdal Kerberos 5 error reporting functions** .

Keytab management

A keytab is a storage for locally stored keys. Heimdal includes keytab support for Kerberos 5 keytabs, Kerberos 4 srvtab, AFS-KeyFile’s, and for storing keys in memory.

Keytabs are used for servers and long-running services.

See also **The keytab handing functions** and **Heimdal Kerberos 5 keytab handling functions** .

Kerberos crypto

Heimdal includes a implementation of the Kerberos crypto framework, all crypto operations. To create a crypto context call **krb5_crypto_init()**.

See also module **Heimdal Kerberos 5 cryptography functions** .

Walkthrough of a sample Kerberos 5 client

This example contains parts of a sample TCP Kerberos 5 clients, if you want a real working client, please look in appl/test directory in the Heimdal distribution.

All Kerberos error-codes that are returned from kerberos functions in this program are passed to krb5_err, that will print a descriptive text of the error code and exit. Graphical programs can convert error-code to a human readable error-string with the krb5_get_error_message() function.

Note that you should not use any Kerberos function before **krb5_init_context()** have completed successfully. That is the reason err() is used when **krb5_init_context()** fails.

First the client needs to call krb5_init_context to initialise the Kerberos 5 library. This is only needed once per thread in the program. If the function returns a non-zero value it indicates that either the Kerberos implementation is failing or it’s disabled on this host.

```
#include <krb5.h>

int
main(int argc, char **argv)
{
        krb5_context context;

        if (krb5_init_context(&context))
                errx (1, ’krb5_context’);
```

Now the client wants to connect to the host at the other end. The preferred way of doing this is using getaddrinfo (for operating system that have this function implemented), since getaddrinfo is neutral to the address type and can use any protocol that is available.

```
        struct addrinfo *ai, *a;
        struct addrinfo hints;
        int error;

        memset (&hints, 0, sizeof(hints));
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_protocol = IPPROTO_TCP;

        error = getaddrinfo (hostname, ’pop3’, &hints, &ai);
        if (error)
                errx (1, ’%s: %s’, hostname, gai_strerror(error));
```

```
for (a = ai; a != NULL; a = a->ai_next) {
        int s;

        s = socket (a->ai_family, a->ai_socktype, a->ai_protocol);
        if (s < 0)
                continue;
        if (connect (s, a->ai_addr, a->ai_addrlen) < 0) {
                warn ('connect(%s)', hostname);
                    close (s);
                    continue;
        }
        freeaddrinfo (ai);
        ai = NULL;
}
if (ai) {
            freeaddrinfo (ai);
            errx ('failed to contact %s', hostname);
}
```

Before authenticating, an authentication context needs to be created. This context keeps all information for one (to be) authenticated connection (see krb5_auth_context).

```
status = krb5_auth_con_init (context, &auth_context);
if (status)
        krb5_err (context, 1, status, 'krb5_auth_con_init');
```

For setting the address in the authentication there is a help function krb5_auth_con_setaddrs_from_fd() that does everything that is needed when given a connected file descriptor to the socket.

```
status = krb5_auth_con_setaddrs_from_fd (context,
                                         auth_context,
                                         &sock);
if (status)
        krb5_err (context, 1, status,
                  'krb5_auth_con_setaddrs_from_fd');
```

The next step is to build a server principal for the service we want to connect to. (See also **krb5_sname_to_principal()**.)

```
status = krb5_sname_to_principal (context,
```

```
                        hostname,
                        service,
                        KRB5_NT_SRV_HST,
                        &server);
    if (status)
        krb5_err (context, 1, status, 'krb5_sname_to_principal');
```

The client principal is not passed to krb5_sendauth() function, this causes the krb5_sendauth() function to try to figure it out itself.

The server program is using the function krb5_recvauth() to receive the Kerberos 5 authenticator.

In this case, mutual authentication will be tried. That means that the server will authenticate to the client. Using mutual authentication is good since it enables the user to verify that they are talking to the right server (a server that knows the key).

If you are using a non-blocking socket you will need to do all work of krb5_sendauth() yourself. Basically you need to send over the authenticator from krb5_mk_req() and, in case of mutual authentication, verifying the result from the server with krb5_rd_rep().

```
    status = krb5_sendauth (context,
                    &auth_context,
                    &sock,
                    VERSION,
                    NULL,
                    server,
                    AP_OPTS_MUTUAL_REQUIRED,
                    NULL,
                    NULL,
                    NULL,
                    NULL,
                    NULL,
                    NULL);
    if (status)
        krb5_err (context, 1, status, 'krb5_sendauth');
```

Once authentication has been performed, it is time to send some data. First we create a krb5_data structure, then we sign it with krb5_mk_safe() using the auth_context that contains the session-key that was exchanged in the krb5_sendauth()/krb5_recvauth() authentication sequence.

```
data.data   = 'hej';
data.length = 3;

krb5_data_zero (&packet);

status = krb5_mk_safe (context,
                       auth_context,
                       &data,
                       &packet,
                       NULL);
if (status)
        krb5_err (context, 1, status, 'krb5_mk_safe');
```

And send it over the network.

```
len = packet.length;
net_len = htonl(len);

if (krb5_net_write (context, &sock, &net_len, 4) != 4)
        err (1, 'krb5_net_write');
if (krb5_net_write (context, &sock, packet.data, len) != len)
        err (1, 'krb5_net_write');
```

To send encrypted (and signed) data krb5_mk_priv() should be used instead. krb5_mk_priv() works the same way as krb5_mk_safe(), with the exception that it encrypts the data in addition to signing it.

```
data.data   = 'hemligt';
data.length = 7;

krb5_data_free (&packet);

status = krb5_mk_priv (context,
                       auth_context,
                       &data,
                       &packet,
                       NULL);
if (status)
        krb5_err (context, 1, status, 'krb5_mk_priv');
```

And send it over the network.

```
len = packet.length;
net_len = htonl(len);

if (krb5_net_write (context, &sock, &net_len, 4) != 4)
        err (1, 'krb5_net_write');
if (krb5_net_write (context, &sock, packet.data, len) != len)
        err (1, 'krb5_net_write');
```

The server is using krb5_rd_safe() and krb5_rd_priv() to verify the signature and decrypt the packet.

Validating a password in an application

See the manual page for krb5_verify_user().

API differences to MIT Kerberos

This section is somewhat disorganised, but so far there is no overall structure to the differences, though some of the have their root in that Heimdal uses an ASN.1 compiler and MIT doesn't.

Principal and realms

Heimdal stores the realm as a krb5_realm, that is a char *. MIT Kerberos uses a krb5_data to store a realm.

In Heimdal krb5_principal doesn't contain the component name_type; it's instead stored in component name.name_type. To get and set the nametype in Heimdal, use **krb5_principal_get_type()** and **krb5_principal_set_type()**.

For more information about principal and realms, see krb5_principal.

Error messages

To get the error string, Heimdal uses krb5_get_error_message(). This is to return custom error messages (like ‘‘Can't find host/datan.example.com@CODE.COM in /etc/krb5.conf.’’ instead of a ‘‘Key table entry not found’’ that error_message returns.

Heimdal uses a threadsafe(r) version of the com_err interface; the global com_err table isn't initialised. Then error_message returns quite a boring error string (just the error code itself).