NAME

libradius - RADIUS client/server library

SYNOPSIS

#include <radlib.h>

*struct rad_handle **
rad_acct_open(*void*);

int
rad_add_server(*struct rad_handle *h*, *const char *host*, *int port*, *const char *secret*, *int timeout*,
int max_tries);

int
rad_add_server_ex(*struct rad_handle *h*, *const char *host*, *int port*, *const char *secret*, *int timeout,*
*int max_tries, int dead_time, struct in_addr *bindto*);

*struct rad_handle **
rad_auth_open(*void*);

void
rad_close(*struct rad_handle *h*);

int
rad_config(*struct rad_handle *h*, *const char *file*);

int
rad_continue_send_request(*struct rad_handle *h*, *int selected, int *fd, struct timeval *tv*);

int
rad_create_request(*struct rad_handle *h*, *int code*);

int
rad_create_response(*struct rad_handle *h*, *int code*);

struct in_addr
rad_cvt_addr(*const void *data*);

uint32_t
rad_cvt_int(*const void *data*);

*char **
rad_cvt_string(*const void *data*, *size_t len*);

int
rad_get_attr(*struct rad_handle *h*, *const void **data*, *size_t *len*);

int
rad_get_vendor_attr(*uint32_t *vendor*, *const void **data*, *size_t *len*);

int
rad_init_send_request(*struct rad_handle *h*, *int *fd*, *struct timeval *tv*);

int
rad_put_addr(*struct rad_handle *h*, *int type*, *struct in_addr addr*);

int
rad_put_attr(*struct rad_handle *h*, *int type*, *const void *data*, *size_t len*);

int
rad_put_int(*struct rad_handle *h*, *int type*, *uint32_t value*);

int
rad_put_string(*struct rad_handle *h*, *int type*, *const char *str*);

int
rad_put_message_authentic(*struct rad_handle *h*);

int
rad_put_vendor_addr(*struct rad_handle *h*, *int vendor*, *int type*, *struct in_addr addr*);

int
rad_put_vendor_attr(*struct rad_handle *h*, *int vendor*, *int type*, *const void *data*, *size_t len*);

int
rad_put_vendor_int(*struct rad_handle *h*, *int vendor*, *int type*, *uint32_t value*);

int
rad_put_vendor_string(*struct rad_handle *h*, *int vendor*, *int type*, *const char *str*);

ssize_t

rad_request_authenticator(*struct rad_handle *h*, *char *buf*, *size_t len*);

int
rad_receive_request(*struct rad_handle *h*);

int
rad_send_request(*struct rad_handle *h*);

int
rad_send_response(*struct rad_handle *h*);

*struct rad_handle **
rad_server_open(*int fd*);

*const char **
rad_server_secret(*struct rad_handle *h*);

void
rad_bind_to(*struct rad_handle *h, in_addr_t addr*);

*u_char **
rad_demangle(*struct rad_handle *h, const void *mangled, size_t mlen*);

*u_char **
rad_demangle_mppe_key(*struct rad_handle *h*, *const void *mangled*, *size_t mlen*, *size_t *len*);

*const char **
rad_strerror(*struct rad_handle *h*);

DESCRIPTION

The **libradius** library implements the Remote Authentication Dial In User Service (RADIUS). RADIUS, defined in RFCs 2865 and 2866, allows clients to perform authentication and accounting by means of network requests to remote servers.

Initialization

To use the library, an application must first call **rad_auth_open**(), **rad_acct_open**() or **rad_server_open**() to obtain a *struct rad_handle **, which provides the context for subsequent operations. The former function is used for RADIUS authentication and the latter is used for RADIUS accounting. Calls to **rad_auth_open**(), **rad_acct_open**() and **rad_server_open**() always succeed unless insufficient virtual memory is available. If the necessary memory cannot be allocated, the functions return NULL. For compatibility with earlier versions of this library, **rad_open**() is provided as a synonym for **rad_auth_open**().

Before issuing any RADIUS requests, the library must be made aware of the servers it can contact. The easiest way to configure the library is to call **rad_config**(). **rad_config**() causes the library to read a configuration file whose format is described in radius.conf(5). The pathname of the configuration file is passed as the *file* argument to **rad_config**(). This argument may also be given as NULL, in which case the standard configuration file */etc/radius.conf* is used. **rad_config**() returns 0 on success, or -1 if an error occurs.

The library can also be configured programmatically by calls to **rad_add_server**() or **rad_add_server_ex**(). **rad_add_server**() is a backward compatible function, implemented via **rad_add_server_ex**(). The *host* parameter specifies the server host, either as a fully qualified domain name or as a dotted-quad IP address in text form. The *port* parameter specifies the UDP port to contact on the server. If *port* is given as 0, the library looks up the ‘radius/udp’ or ‘radacct/udp’ service in the network services(5) database, and uses the port found there. If no entry is found, the library uses the standard RADIUS ports, 1812 for authentication and 1813 for accounting. The shared secret for the server host is passed to the *secret* parameter. It may be any NUL-terminated string of bytes. The RADIUS protocol ignores all but the leading 128 bytes of the shared secret. The timeout for receiving replies from the server is passed to the *timeout* parameter, in units of seconds. The maximum number of repeated requests to make before giving up is passed into the *max_tries* parameter. Time interval in seconds when the server will not be requested if it is marked as dead (did not answer on the last try) set with *dead_time* parameter. *bindto* parameter is an IP address on the multihomed host that is used as a source address for all requests. **rad_add_server**() returns 0 on success, or -1 if an error occurs.

rad_add_server() or **rad_add_server_ex**() may be called multiple times, and they may be used together with **rad_config**(). At most 10 servers may be specified. When multiple servers are given, they are tried in round-robin fashion until a valid response is received, or until each server’s *max_tries* limit has been reached.

Creating a RADIUS Request

A RADIUS request consists of a code specifying the kind of request, and zero or more attributes which provide additional information. To begin constructing a new request, call **rad_create_request**(). In addition to the usual *struct rad_handle* *, this function takes a *code* parameter which specifies the type of the request. Most often this will be RAD_ACCESS_REQUEST. **rad_create_request**() returns 0 on success, or -1 on if an error occurs.

After the request has been created with **rad_create_request**(), attributes can be attached to it. This is done through calls to **rad_put_addr**(), **rad_put_int**(), and **rad_put_string**(). Each accepts a *type* parameter identifying the attribute, and a value which may be an Internet address, an integer, or a NUL-terminated string, respectively. Alternatively, **rad_put_vendor_addr**(), **rad_put_vendor_int**() or **rad_put_vendor_string**() may be used to specify vendor specific attributes. Vendor specific definitions may be found in <*radlib_vs.h*>

The library also provides a function **rad_put_attr**() which can be used to supply a raw, uninterpreted attribute. The *data* argument points to an array of bytes, and the *len* argument specifies its length.

It is possible adding the Message-Authenticator to the request. This is an HMAC-MD5 hash of the entire Access-Request packet (see RFC 3579). This attribute must be present in any packet that includes an EAP-Message attribute. It can be added by using the **rad_put_message_authentic**() function. The **libradius** library calculates the HMAC-MD5 hash implicitly before sending the request. If the Message-Authenticator was found inside the response packet, then the packet is silently dropped, if the validation failed. In order to get this feature, the library should be compiled with OpenSSL support.

The **rad_put_X**() functions return 0 on success, or -1 if an error occurs.

Sending the Request and Receiving the Response

After the RADIUS request has been constructed, it is sent either by means of **rad_send_request**() or by a combination of calls to **rad_init_send_request**() and **rad_continue_send_request**().

The **rad_send_request**() function sends the request and waits for a valid reply, retrying the defined servers in round-robin fashion as necessary. If a valid response is received, **rad_send_request**() returns the RADIUS code which specifies the type of the response. This will typically be RAD_ACCESS_ACCEPT, RAD_ACCESS_REJECT, or RAD_ACCESS_CHALLENGE. If no valid response is received, **rad_send_request**() returns -1.

As an alternative, if you do not wish to block waiting for a response, **rad_init_send_request**() and **rad_continue_send_request**() may be used instead. If a reply is received from the RADIUS server or a timeout occurs, these functions return a value as described for **rad_send_request**(). Otherwise, a value of zero is returned and the values pointed to by *fd* and *tv* are set to the descriptor and timeout that should be passed to select(2).

rad_init_send_request() must be called first, followed by repeated calls to **rad_continue_send_request**() as long as a return value of zero is given. Between each call, the application should call select(2), passing **fd* as a read descriptor and timing out after the interval specified by *tv*. When select(2) returns, **rad_continue_send_request**() should be called with *selected* set to a non-zero value if select(2) indicated that the descriptor is readable.

Like RADIUS requests, each response may contain zero or more attributes. After a response has been received successfully by **rad_send_request**() or **rad_continue_send_request**(), its attributes can be extracted one by one using **rad_get_attr**(). Each time **rad_get_attr**() is called, it gets the next attribute from the current response, and stores a pointer to the data and the length of the data via the reference parameters *data* and *len*, respectively. Note that the data resides in the response itself, and must not be modified. A successful call to **rad_get_attr**() returns the RADIUS attribute type. If no more attributes remain in the current response, **rad_get_attr**() returns 0. If an error such as a malformed attribute is detected, -1 is returned.

If **rad_get_attr**() returns RAD_VENDOR_SPECIFIC, **rad_get_vendor_attr**() may be called to determine the vendor. The vendor specific RADIUS attribute type is returned. The reference parameters *data* and *len* (as returned from **rad_get_attr**()) are passed to **rad_get_vendor_attr**(), and are adjusted to point to the vendor specific attribute data.

The common types of attributes can be decoded using **rad_cvt_addr**(), **rad_cvt_int**(), and **rad_cvt_string**(). These functions accept a pointer to the attribute data, which should have been obtained using **rad_get_attr**() and optionally **rad_get_vendor_attr**(). In the case of **rad_cvt_string**(), the length *len* must also be given. These functions interpret the attribute as an Internet address, an integer, or a string, respectively, and return its value. **rad_cvt_string**() returns its value as a NUL-terminated string in dynamically allocated memory. The application should free the string using free(3) when it is no longer needed.

If insufficient virtual memory is available, **rad_cvt_string**() returns NULL. **rad_cvt_addr**() and **rad_cvt_int**() cannot fail.

The **rad_request_authenticator**() function may be used to obtain the Request-Authenticator attribute value associated with the current RADIUS server according to the supplied rad_handle. The target buffer *buf* of length *len* must be supplied and should be at least 16 bytes. The return value is the number of bytes written to *buf* or -1 to indicate that *len* was not large enough.

The **rad_server_secret**() returns the secret shared with the current RADIUS server according to the supplied rad_handle.

The **rad_bind_to**() assigns a source address for all requests to the current RADIUS server.

The **rad_demangle**() function demangles attributes containing passwords and MS-CHAPv1 MPPE-Keys. The return value is NULL on failure, or the plaintext attribute. This value should be freed using free(3) when it is no longer needed.

The **rad_demangle_mppe_key**() function demangles the send- and recv-keys when using MPPE (see RFC 2548). The return value is NULL on failure, or the plaintext attribute. This value should be freed using free(3) when it is no longer needed.

Obtaining Error Messages

Those functions which accept a *struct rad_handle ** argument record an error message if they fail. The error message can be retrieved by calling **rad_strerror**(). The message text is overwritten on each new error for the given *struct rad_handle **. Thus the message must be copied if it is to be preserved through subsequent library calls using the same handle.

Cleanup

To free the resources used by the RADIUS library, call **rad_close**().

Server operation

Server mode operates much alike to client mode, except packet send and receive steps are swapped. To operate as server you should obtain server context with **rad_server_open**() function, passing opened and bound UDP socket file descriptor as argument. You should define allowed clients and their secrets using **rad_add_server**() function. port, timeout and max_tries arguments are ignored in server mode. You should call **rad_receive_request**() function to receive request from client. If you do not want to block on socket read, you are free to use any poll(), select() or non-blocking sockets for the socket. Received request can be parsed with same parsing functions as for client. To respond to the request you should call **rad_create_response**() and fill response content with same packet writing functions as for client. When packet is ready, it should be sent with **rad_send_response**().

RETURN VALUES

The following functions return a non-negative value on success. If they detect an error, they return -1 and record an error message which can be retrieved using **rad_strerror**().

rad_add_server()
rad_config()
rad_create_request()
rad_create_response()
rad_get_attr()
rad_put_addr()
rad_put_attr()
rad_put_int()
rad_put_string()
rad_put_message_authentic()
rad_init_send_request()
rad_continue_send_request()
rad_send_request()
rad_send_response()

The following functions return a non-NULL pointer on success. If they are unable to allocate sufficient virtual memory, they return NULL, without recording an error message.

rad_acct_open()
rad_auth_open()
rad_server_open()
rad_cvt_string()

The following functions return a non-NULL pointer on success. If they fail, they return NULL, with recording an error message.

rad_demangle()
rad_demangle_mppe_key()

FILES

/etc/radius.conf

SEE ALSO

radius.conf(5)

C. Rigney, et al, *Remote Authentication Dial In User Service (RADIUS)*, RFC 2865.

C. Rigney, *RADIUS Accounting*, RFC 2866.

G. Zorn, *Microsoft Vendor-specific RADIUS attributes*, RFC 2548.

C. Rigney, et al, *RADIUS extensions*, RFC 2869.

AUTHORS

This software was originally written by John Polstra, and donated to the FreeBSD project by Juniper Networks, Inc. Oleg Semyonov subsequently added the ability to perform RADIUS accounting. Later additions and changes by Michael Bretterklieber. Server mode support was added by Alexander Motin.