cisco Systems

Document Number	ENG-####
Revision	1.30
Author	Dave Carrel
Project Manager	Project manager name

TACACS+

API Definition

Project Headline

This is the TACACS+ API definition. TACACS+ is an access control protocol used in cisco products. It provides the services of Authentication, Authorization and Accounting

Reviewers

Department	Name	Acceptance Date
Engineering	Name (Title)	Date
Engineering	Name (Title)	Date
Engineering	Name (Title)	Date

Modification History

Rev	Date	Originator	Comment
1.28	26 Jan 95	Dave Carrel	imported from ascii version
1.29	31 Jan 95	Dave Carrel	changed server_msg and data to be pointers and defined memory handling for them
1.30	2 Feb 95	Dave Carrel	cleaning, nothing more

This document is a product of cisco Systems security development. It is intended to be published as an informational document describing the programming interface to our TACACS+ server. TACACS+ is a protocol that is implemented on cisco products as well as others.

Abstract

One of the goals of this document is to define a standard method for managing dissimilar Network Access Servers (NAS's) from a single set of management services. In particular, the services of concern are the authentication, authorization and accounting services. To that end, discussion has focused on a unified protocol through which a NAS would make authentication, authorization and accounting requests to the NAS server. This document makes a case for moving these efforts towards standardization of a NAS server API that would allow for a single set of network services but would allow each NAS and it's NAS server to communicate using their own protocol.

1.0 Introduction

The nasreq-requirements draft [2] defined a set of requirements for a class of devices referred to as Network Access Servers (NAS's). The term "Network Access Server" is used instead of the more conventional term "Terminal Server" as it more accurately describes the devices of interest.

A Network Access Server (NAS) is a router or terminal server which provides temporary access to a network. It can provide access for both traditional "dumb terminals" and terminal emulators as well as workstations, PC's or routers utilizing a serial line framing protocol such as PPP or SLIP. Thus a NAS can provide connections to a single user, to a network or subnetwork, or to interconnected networks. The entities that use the NAS to connect to a network are called Network Access Clients (NACs).

In addition a NAS helper was called for that interfaces the NAS to the authentication, authorization and accounting services. This model allowed for complex and varying combinations of different services. The NAS helper design used a protocol to allow dissimilar NASs to communicate with a NAS helper. While this design may be ideal, it does not address very well the realities of the current day. These are that every manufacturer has implemented their own protocol for centralized services and in most cases, this protocol reflects the design of the NAS itself. Changing the protocol is often not simple without changing the nature of the NAS. This is not likely to happen as quickly as some would like.

To improve on this situation, this document presents an API that is generalized to allow for integration of different NAS helper servers with the common services that are needed to support all NASs. The task of maintaining multiple NAS servers is assumed small by comparison to the task of maintaining several copies of the network services databases. The API consists of three services: authentication, authorization and accounting. This document does not attempt to solve the general case APIs for any of the three services, but presents specific solutions for the NAS case.

The TACACS+ API was designed in parallel to the TACACS+ protocol specification [1]. Both provide complementary services, but neither depends on the other. The TACACS+ protocol is designed to provide secure, reliable delivery of authentication, authorization and accounting services.

2.0 The TACACS+ API.

The TACACS+ API is presented here as a general purpose solution for integrating NAS servers with authentication, authorization and accounting services. The API design is intended to accommodate any NAS server design. A major factor in the design was to recognize that most other NAS servers will not operate the same way as TACACS+. Another goal of the design was to not limit the API to currently understood services, but to create a general interface that would extend. The hope is that services such as authorization will be addressed in a global sense by other parties and that this interface will provide the NAS server access to that work.

This API provides multiple, run-time selectable, interfaces for authentication and a single interface each for authorization and accounting. The API also defines the interface to a function for choosing the appropriate authentication service based on information about the authenticating user.

3.0 The Authentication API

The Authentication API consists of two functions: a “chooser” function and one or more “authentication” functions. Authentication works generally as follows: a “chooser” function is called which chooses an “authentication” function. This authentication function is then called upon to do the actual authentication.

Both the chooser function and the authentication function may query the NAS for more information by returning special status values. When this happens, TACACS+ will query the NAS for more information, update its data structures, and then call the functions again. At all times, if the chooser or authentication functions return information in the `server_msg` and/or the `server_data` fields, they MUST allocate the memory for those fields and the calling function MUST free that memory.

The chooser function, `choose_authen()`, takes a pointer to an `authen_data` structure and a pointer to an `authen_type` structure. The `authen_data` structure identifies the user that is requesting authentication. On successful completion, the `authen_type` structure will indicate `choose_authen()`’s response by containing a pointer to an authentication function.

The functions and structures are defined by the include file below.

`choose_authen()` may be called with a NULL username in which case it will return 0 if it managed to choose an authentication function anyway. A return of -1 indicates failure. A return value of 1 indicates that `choose_authen()` requires a a username which was not supplied. In this case, `choose_authen()` may optionally fill in `authen_data.server_msg` with an appropriate prompt for acquiring the username (the caller will free the storage associated with this message). The prompt is sent to the NAS and when a reply is received, `choose_authen()` should be called again with the username field filled in the identity structure.

`choose_authen()` may also be called with `authen_type.authen_type` already filled in (i.e. set to a non-zero value). `choose_authen()` must return 0 only if it chose an authentication function corresponding to the requested `authen_type`. A return value of 2 indicates that the indicated `authen_type` is unacceptable. If `choose_authen()` returns in this way, it may specify a list of valid `authen_type`’s by placing the single byte values in the `authen_data.server_data` and setting `authen_data.server_dlen`. The storage for `authen_data.server_data` must be allocated by the `choose_authen()` function and the caller will free that storage.

The authentication function returned by a successful call to `choose_authen()` may also need to be called several times during authentication. The first time it is called, the method_data pointer must be NULL. The authentication function may then use this field to point to private data containing any state information it requires.

On the first call, the authentication function should either initialize itself (allocate the `method_data`) and return the first server message(s) and/or data, or it may terminate the authentication sequence. Subsequent calls are denoted by a non-NULL `method_data` pointer and messages and data are passed back and forth. If an authentication function does not need to use the `method_data`, it may choose to set the pointer to a value that it understands, such as -1. This will be the only indication that the function has, as to whether this is an initial call or a follow-up call. It is the responsibility of the authentication function to free this data. The authentication function is guaranteed to either be able to end the session itself, or to be passed an abort flag (see below) in which case it MUST end the session.

When the authentication session is completed, the authentication function returns and MUST set `authen_data.status` to `AUTHEN_STATUS_SUCCEED`, `AUTHEN_STATUS_FAIL` or `AUTHEN_STATUS_ERROR`. At this point, it MUST free any private data stored in the `method_data`.

When the authentication function returns but is at an intermediate stage in the authentication sequence it sets the `authen_data.status` to `AUTHEN_STATUS_GETDATA`, `AUTHEN_STATUS_GETUSER` or `AUTHEN_STATUS_GETPASS` which indicates that more data is required. The `server_msg` field may contain a string that the NAS may present to the user as a prompt for further input. The calling function MUST free the memory associated with the `server_msg` and `server_data` fields. When a reply is received from the NAS, the authentication function is called again with the `client_msg` and `client_data` fields of `authen_data` filled in as is appropriate.

`AUTHEN_STATUS_GETPASS` and `AUTHEN_STATUS_GETUSER` are special cases of the `AUTHEN_STATUS_GETDATA` and should be treated exactly the same in the TACACS+ server. The only significance is that these messages allow the NAS to understand the context of the data that it is requesting. Making use if that information is a NAS specific detail but possible uses include storing usernames for further authorization or accounting purposes.

The `authen_data.flags` field is used for incoming and outgoing flags. The flags are bitmapped and it is the responsibility of both the authentication function and the protocol engine to clear the flags that are not being used. Incoming flags include the abort flag which tells the authentication function to terminate the session and clean up. When the authentication function is called, if the abort flag is set, the `authen_data.client_data` MAY contain an ascii message from the client explaining the abort. No return messages may be sent after the abort flag is passed in. For outgoing flags, there is the no-echo flag. This flag is used along with the statuses `AUTHEN_STATUS_GET*` to allow the authentication function to specify that the user response should not be echoed as it is typed.

The `authen_data.action` can describe three possible actions. `AUTHEN_ACTION_LOGIN` and `AUTHEN_ACTION_CHPASS` describe a login authentication request and a password change request respectively. Both of these actions are fufilled with the standard `REPLY` and `CONTINUE` messages until the server decides to finish the session. `AUTHEN_ACTION_SENDPASS` is a request to return the cleartext password associated with a user. This action describes an authentication mechanism where the server is not making the final pass or fail decision, but rather is providing the necessary password so that the NAS can perform this action. `AUTHEN_ACTION_SENDPASS` is used when the NAS is performing an authentication sequence such as CHAP or ARAP authentication. The returned password is placed in the `server_data` field. (The calling function MUST free all memory associated with the `server_data` field.)

```
#ifndef _AUTHEN_H_
#define _AUTHEN_H_

/*
 * This structure describes an authentication method.
 *   authen_name     contains the name of the authentication method.
 *   authen_func     is a pointer to the authentication function.
 *   authen_type     numeric value of authentication method
 */
struct authen_type {
     char    authen_name[128];
     int     authen_func(struct authen_data *, int);
     int     authen_type;
};
```

```
/*
 * Possible values for authen_type.authen_format
 */
#define AUTHEN_TYPE_ASCII    1
#define AUTHEN_TYPE_PAP      2
#define AUTHEN_TYPE_CHAP     3
#define AUTHEN_TYPE_ARAP     4

/*
 * This structure describes a principal that is to be authenticated.
 *   username        is the principals name (ASCII, null terminated)
 *   NAS_name        is the name of the NAS where the user is
 *   NAS_port        is the port on the NAS where the user is
 *   NAC_address     is the remote user location.  This may be
 *                   a remote IP address or a caller-ID or ...
 *   priv_lvl        user's requested privilege level.
 */
struct identity {
     char    username[32];
     char    NAS_name[32];
     char    NAS_port[32];
     char    NAC_address[64];
     int     priv_lvl;
};

/*
 * Possible values for priv_lvl.
 */
#define AUTHEN_PRIV_MAX      0x0f    /* Maximum level allowed */
#define AUTHEN_PRIV_ROOT     0x0f    /* Super-user privilege */
#define AUTHEN_PRIV_OPR      0x0f    /* Operator privilege */
#define AUTHEN_PRIV_CONF     0x0f    /* Configuration privilege */
#define AUTHEN_PRIV_USER     0x01    /* Standard user privilege level */
#define AUTHEN_PRIV_MIN      0x00    /* Minimum level allowed */
```

```
/*
 * The authen_data structure is the data structure for passing
 * information to and from the authentication function
 * (authen_type.authen_func).
 */

struct authen_data {
     struct identity        *NAS_id;          /* user identity */
     char                   *server_msg;      /* output msg */
     int                     server_dlen;     /* output data length */
     char                   *server_data;     /* output data */
     char                   *client_msg;      /* input msg */
     int                     client_dlen;     /* input data length */
     char                   *client_data;     /* input data */
     void                   *method_data;     /* opaque method data */
     int                     action;          /* what's to be done */
     int                     service;         /* calling service */
     int                     status;          /* Authen status */
     int                     type;            /* Authen type */
     uchar                   flag;            /* in and out flags */
};

/*
 * Possible values for authen_data.action
 */
#define AUTHEN_ACTION_LOGIN          0x01
#define AUTHEN_ACTION_CHPASS         0x02
#define AUTHEN_ACTION_SENDPASS       0x03

/*
 * Possible values for authen_data.service
 */
#define AUTHEN_SERVICE_LOGIN         0x01
#define AUTHEN_SERVICE_ENABLE        0x02
#define AUTHEN_SERVICE_PPP           0x03
#define AUTHEN_SERVICE_ARAP          0x04
#define AUTHEN_SERVICE_PT            0x05
#define AUTHEN_SERVICE_RCMD          0x06
#define AUTHEN_SERVICE_X25           0x07
```

```
/*
 * Possible values for authen_data.type
 */
#define AUTHEN_TYPE_ASCII            0x01
#define AUTHEN_TYPE_PAP              0x02
#define AUTHEN_TYPE_CHAP             0x03
#define AUTHEN_TYPE_ARAP             0x04
/*
 * Possible values for authen_data.status
 */
#define AUTHEN_STATUS_SUCCEED        0x01
#define AUTHEN_STATUS_FAIL           0x02
#define AUTHEN_STATUS_GETDATA        0x03
#define AUTHEN_STATUS_GETUSER        0x04
#define AUTHEN_STATUS_GETPASS        0x05
#define AUTHEN_STATUS_ERROR          0x07    /* Server error */

/*
 * Bitmapped values for the input flags.
 */
#define AUTHEN_FLAG_IN_ABORT         0x01
/*
 * Bitmapped values for the output flags.
 */
#define AUTHEN_FLAG_OUT_NOECHO       0x01

/*
 * This function chooses the authentication method to use based on
 * the information in the authen_data structure.
 *
 * Returns:  0       on success
 *           1       failed because username is missing.
 *           2       failed because authen_type.authen_type is bad
 *          -1       on failure
 *
 * Input:    authen_data
 *           authen_type         To return function name and pointer.
 */
extern int
choose_authen(struct authen_data *, struct authen_type *);
```

```
/*
 * This function is the authentication entry point for any available
 * authentication method.
 *
 * Returns:  0       Success (authentication pass OR fail.)
 *          -1       Fail (not authentication failure.)
 *
 * Input:    authen_data
 */
/* extern int
 * (authen_type.authen_func)(struct authen_data *);
 */

#endif /* _AUTHEN_H_ */
```

4.0 The Authorization API

The Authorization API defines a single function. This function is called once per authorization. It takes information about the user and information about the service to be executed and responds with a status of `AUTHOR_STATUS_FAIL`, `AUTHOR_STATUS_PASS_ADD`, `AUTHOR_STATUS_PASS_REPL` or `AUTHOR_STATUS_ERROR`. The authorization API uses the same identity structure as the authentication API. The information about the service is in the form of an arbitrarily long list of attribute-value pairs. The attributes describe the service being authorized and allow for security restriction and per user customization. The function returns the following status values:

If the status is `AUTHOR_STATUS_FAIL`, the service is denied unconditionally. The function MAY return a message that is to be presented to the user.

If the status is `AUTHOR_STATUS_PASS_ADD`, the command is allowed and the arguments returned in the author_data.output_args will be considered additional attributes to be added to original set of attributes. To send passing status with no modifications to the original set of attributes, the response should be `AUTHOR_STATUS_PASS_ADD` with zero output arguments.

If the status is `AUTHOR_STATUS_PASS_REPL`, the command is allowed but some of the original arguments were either denied or modified. A complete replacement list of arguments must be returned in `author_data.output_args`.

If the status is `AUTHOR_STATUS_ERROR`, then the authorization is incomplete and the function may return an administrative message.

For all status values, an user message may be returned in the `msg` field.

As not all NAS's start services the same way, the authorization engine must understand the proper attribute-value pairs for each type of NAS that it supports. The interface is defined by the following include file.

```
#ifndef _AUTHOR_H_
#define _AUTHOR_H_

/*
 * This structure is the data structure for passing information to
 * and from the authorization function do_author().
 */
struct author_data {
     struct identity    *id;             /* user id */
     int                 authen_meth;    /* authentication method */
     int                 authen_type;    /* defined in authen.h */
     int                 service;        /* defined in authen.h */
     char               *msg;            /* return message */
     char               *admin_msg;      /* return msg for admin */
     int                 status;         /* return status */
     int                 num_in_args     /* number of input args */
     char              **input_args;     /* service arguments */
     int                 num_out_args    /* number of output args */
     char              **output_args;    /* service arguments */
};

/*
 * Possible status values for author_data.authen_meth
 */
#define AUTHEN_METH_NONE             0x01
#define AUTHEN_METH_KRB5             0x02
#define AUTHEN_METH_LINE             0x03
#define AUTHEN_METH_ENABLE           0x04
#define AUTHEN_METH_LOCAL            0x05
#define AUTHEN_METH_TACACSPLUS       0x06
#define AUTHEN_METH_RCMD             0x20

/*
 * Possible status values for author_data.status
 */
#define AUTHOR_STATUS_PASS_ADD       0x01
#define AUTHOR_STATUS_PASS_REPL      0x02
#define AUTHOR_STATUS_FAIL           0x10
#define AUTHOR_STATUS_ERROR          0x11
```

```
/*
 * This function is the authorization entry point.
 *
 * Returns:  0       Success. Return status is valid
 *          -1       Error
 *
 * Input:    author_data     describes the principal and
 *                           provides current authorization info.
 */
extern int
do_author(struct author_data *);

#endif /* _AUTHOR_H_ */
```

The `msg` and `admin_msg` fields are only used to return ascii messages from the `do_author()` function. The `msg` field may be used at any time and contains a message to be presented to the user. The `admin_msg` is only used when `do_author()` returns an `AUTHOR_STATUS_ERROR` in `author_data.status` and it contains a message to be displayed on a administrative console or log on the client. The `do_author()` function MUST allocate any memory associated with the `msg` and `admin_msg` fields, or set them to zero. The calling function MUST free this memory.

It is preferable that `do_author()` return 0 and set `author_data.status` to `AUTHOR_STATUS_ERROR`, than to return -1. If `do_author()` returns -1 then the contents of `author_data` are considered invalid and are not used.

`input_args` and `output_args` are arrays of NULL terminated character strings. The array is not NULL terminated because there is a count value for each array. The `do_author()` function will have to allocate memory for the `output_args` unless `num_out_args` is zero, in which case memory is not allocated for any argument strings nor for the pointer array, and `output_args` MUST also set to zero. The `do_author()` function MUST allocate the `output_args` as a single memory allocation to store the array of pointers, and then separate memory allocations to store the argument strings. The server which calls `do_author()` MUST de-allocate this memory.

5.0 The Accounting API

The Accounting API contains only one function. This function takes information about the user and the service and writes an accounting/audit record. The record contains information about the user The function return value indicates whether or not the record was successfully recorded. If the function returns a failure, it may include an ascii message for the user and one for the administrator.

The start time, stop time and byte and packet counts are included as arguments. Not all clients and all accounting records will contain all of these. The exact attributes for describing this accounting specific data are defined in the TACACS+ protocol specification [1] in the accounting section.

The interface is described by this include file.

```
#ifndef _ACCT_H_
#define _ACCT_H_

struct acct_rec {
     int                  acct_type;      /* Start, stop or update */
     struct identity     *identity;
     int                  authen_meth;    /* defined in author.h */
     int                  authen_type;    /* defined in authen.h */
     int                  service;        /* defined in authen.h */
     char                *msg;            /* return message */
     char                *admin_msg;      /* return msg for admin */
     int                  num_args;       /* number of args */
     char                *args;           /* arguments */
};

/*
 * Possible values for acct_rec.acct_type
 */
#define ACCT_TYPE_START      0x01
#define ACCT_TYPE_STOP       0x02
#define ACCT_TYPE_UPDATE     0x03

/*
 * This function writes an accounting/auditing record containing
 * the information in the acct_rec structure.
 *
 * Returns:  0       on success
 *          -1       on failure
 *
 * Input:    acct_rec        All the information to be recorded.
 */
extern int
do_acct(struct acct_rec *);

#endif /* _ACCT_H_ */
```

The `msg` and `admin_msg` fields are only used to return ascii messages from the `do_acct()` function. Both fields are only used when `do_acct()` returns -1. The `msg` contains a message to be presented to the user. The `admin_msg` contains a message to be displayed on a administrative console or log on the client. The `do_acct()` function MUST allocate any memory associated with the `msg` and `admin_msg` fields, or set them to zero. The calling function MUST free this memory.

`args` is a non NULL terminated array of pointers to NULL terminated strings.

6.0 Security Considerations

Security is a fundamental part of this work. The services accessed through this API are some of the fundamental core security components of any system. This API does not provide security in itself. While access to the API does not open any security weaknesses, it's use does not guarantee any security. Any program implementing this API must ensure that it does so in a secure way. Likewise, any integration of services with this API must guarantee security. To put it concisely: "It's all in the implementation and administration."

7.0 References

[1] Carrel, D., Grant, L.,
"The TACACS+ protocol specification."

[2] Vollbrecht, J., Rubens, A., McGregor, G., Blunk, L., Conto, R.,
"Network Access Server Proposed Requirements Document",
draft-ietf-nasreq-nasrequirements-01.txt, Merit Network Inc.,
July 1993 (expired January 1994).

[3] Finseth, C.,
RFC 1492, "An Access Control Protocol, Sometimes Called TACACS",
July 1993.

8.0 Acknowledgments

Some of the introductory text was borrowed from [2].

Many of the ideas and motivations for this came from discussions on the NASREQ mailing list.

9.0 Authors' Addresses

Questions about this memo can also be directed to:

David Carrel
Security Development
cisco Systems, Inc., 170 W. Tasman Dr., San Jose, CA 95134-1706
Phone: (408) 526-5207
Email: carrel@cisco.com

Lol Grant
cisco Systems, Inc., 170 W. Tasman Dr., San Jose, CA 95134-1706
Phone: (408) 526-8171
Email: lol@cisco.com