

TAC_PLUS User's Guide v2.1

Copyright (c) 1995 by Cisco systems, Inc.
All rights reserved.

Please NOTE: None of the TACACS code available here comes with any warranty or support.

Definitions and Terms

NAS --- A Network Access Server i.e. a Cisco box, or anything else which makes tacacs+ authentication and authorization requests, or sends accounting packets.

Daemon -- a program, usually running on a Unix host, which services network requests for authentication and authorization, verifies identities, grants or denies authorizations, and logs accounting records.

passwd(5) files -- files conforming to Unix password style format, as documented in section 5 of the Unix manuals.

AV pairs -- strings of text in the form attribute=value sent between a NAS and a tacacs+ daemon.

Since a NAS is often referred to as a server, and a daemon is also often referred to as a server, the term "server" has been avoided here in favor of the less ambiguous terms "NAS" and "Daemon".

Note that there are now at least 3 versions of authentication protocol that people commonly refer to as "TACACS". The first is ordinary tacacs, which was the first one offered on Cisco boxes, and has been in use for many years. The second is an extension to the first, commonly called Extended Tacacs or XTACACS, introduced in 1990. The third one is Tacacs+ (or tac_plus) which is what is documented here. Tacacs+ is not compatible with any previous versions of tacacs.

In addition to the 3 versions of tacacs running on Cisco boxes, the fact that we distribute the source code to the daemon has meant that additional versions of tacacs daemons have been produced by people who have made modifications to our source code.

UPGRADING FROM VERSION 1.0

The syntax of the configuration file has changed slightly between V1.0 and in V1.1 or later. The keyword "password" has been changed to

"login" and all passwords now indicate what type they are (e.g. either cleartext or des).

The easiest way to convert a 1.0 configuration file is to do some simple global edits in the following order:

1). Change all occurrences of the strings

```
password = cleartext
```

into

```
login = cleartext
```

2). Change all occurrences of

```
arap =
```

into

```
arap = cleartext
```

3). Change all occurrences of

```
chap =
```

into

```
chap = cleartext
```

4). Change all remaining occurrences of

```
password =
```

into

```
login = des
```

That's all that is required if you are upgrading from V1.0 to V1.1.

No configuration file changes are required to upgrade from V1.1, although the "default user = permit" directive is deprecated in favour of "default authorization = permit".

BUILDING TAC_PLUS

Tac_plus is known to build and run on the following platforms:

AIXv3.2 (using bsdcc, see /usr/lpp/box/bsdport on your system)

HP/UX A.09.01 using -DHPUX

i86 Solaris 2.4 (SUNOS 5.4), using SUNpro SW2.0.1 & -DSDOLARIS

Sun4 Solaris 2.4 using SUNpro SC3.0.1 and -DSDOLARIS

SUNOS 4.1.2 sparc-2

SUNOS 4.1.3

MIPS R3K SGI IRIX 4.05F (using -DMIPS)

BSDI BSD/386 1.1
FREEBSD 2.0-RELEASE
LINUX 1.2.8.

To build `tac_plus`, untar the tarfile distribution into a clean directory e.g.

```
tar -xvf tac_plus.2.1.tar
```

then edit the top of the Makefile to select the appropriate defaults for your system. Then type

```
make tac_plus
```

The default version can authenticate using its internal database, `s/key` or `passwd(5)` style files. Authorization is done using the internal database or via calls to external programs, which the administrator configures.

To use S/KEY, you must obtain and build the `s/key` library (`libskey.a`) yourself. You can then compile in S/KEY support per the instructions for S/KEY in the Makefile. I got my S/KEY code from `crimelab.com` [`198.64.127.1`] in directory: `/pub/skey`.

Should you need them, there are routines for accessing password files (`getpwnam`, `setpwent`, `endpwent`, `setpwfile`) in `pw.c`.

CONFIGURING TAC_PLUS

`Tac_plus` is configured via a configuration file. You can create a configuration file from scratch or, if you have `passwd(5)` and supplementary files from earlier versions of `tacacs`, you can convert these to configuration file format by running the supplied perl script `convert.pl`.

CONVERTING EXISTING PASSWD(5) FILES

To convert an existing `passwd(5)` file e.g. one used with an older version of `tacacs`, use the `convert.pl` perl script as follows:

```
convert.pl <passwd file> [-g] [<supplementary file>]
```

- 1). If you have no supplementary file, simply omit it.
- 2). If the `groupid` field of your `passwd` file does NOT represent a valid `acl` number (e.g if it's really a `unix passwd` file this field is a group id, not an `acl` number), just omit the `-g` flag.

The rest of this document assumes that are configuring `tac_plus` from scratch.

CONFIGURING TAC_PLUS FROM SCRATCH

A configuration file consists of some top-level directives for setting defaults and for setting up the encryption key, followed by a declaration for each user and group you want to configure. Within each user or group declaration, there are declarations for authenticating and authorizing operations for that user.

1). Configuring the encryption key

If you want `tac_plus` to encrypt its packets (and you almost certainly **DO** want this, as there can be usernames and passwords contained in these packets), then you must specify an encryption key in the configuration file. The identical key must also be configured on any NAS which communicates with `tac_plus`.

This is done using the statement

```
key = "your key here"
```

NOTE: You only need the double quotes on the daemon if your key contains spaces.

Confusingly, even if your key does contain spaces, you should never use double quotes when you configure the matching password on the NAS.

During debugging, it may be convenient to temporarily switch off encryption by not specifying any key. Be careful to remember to switch encryption back on again after you've finished debugging.

The current code does not support host-specific keys (left as an exercise to the reader).

On the NAS, you also need to configure the **same** key. Do this by issuing:

```
aaa new-model
tacacs-server key <your key here>
```

COMMENTS IN CONFIGURATION FILES

Comments can appear anywhere in the configuration file, starting with the `#` character and extending to the end of the current line. Should you need to disable this special meaning of the `#` character, e.g. if you have a password containing a `#` character, simply enclose the string containing it within double quotes.

CONFIGURING USERS AND GROUPS

Each user may belong to a group (but only one group). Each group may

in turn belong to one other group and so on ad infinitum.

Users and groups are declared as follows. Here we declare two users "fred" and "lily", and two groups, "admin" and "staff".

Fred is a member of group "admin", and group "admin" is in turn a member of group "staff". Lily is not a member of any group.

```

user = lily {
    # user lily is not a member of any group
    # and has nothing else configured as yet
}

user = fred {
    # fred is a member of group admin
    member = admin
}

group = admin {
    # group admin is a member of group staff
    member = staff
}

group = staff {
    # group staff is not a member of any group
}

```

RECURSION AND GROUPS

In general, when the daemon looks up values e.g. passwords, it will look first to see if the user has it defined e.g. if the user has her own password. If not, it looks to see if she belongs to a group and if so, whether the group has a password defined. If not, this process continues through the hierarchy of groups (a group can be a member of another group) until a value is found, or there are no more groups.

This recursive process occurs for lookups of expiration dates, for arap and chap "secrets", and also for authorization parameters (see later).

A typical configuration technique is thus to place users into groups and specify as many groupwide characteristics in the group declaration as possible. Then, individual user declarations can be used to override the group settings for selected users as needed.

CONFIGURING USER AUTHENTICATION

User Authentication can be specified separately for ARAP, CHAP, and normal logins. In addition, a user global authentication method can be given that will be used if a per-protocol method is not specified.

ARAP, CHAP, and global user authentication must be given in clear text.

The following assigns the user mary three different passwords for ARAP, CHAP and normal login respectively:

```

user = mary {
    arap = cleartext "arap password"
    chap = cleartext "chap password"
    login = des XQj4892fjk
}

```

The following assigns the user agnes a single password for all three types of login:

```

user = agnes {
    global = cleartext "Agnes global password"
}

```

There are 3 ways to authenticate a user for login.

1). You can include a DES (or cleartext) password for a user or for a group that s/he is a member of, viz:

```

user = joe {
    member = admin
    # this is lily's encrypted DES password. It overrides the
admin
    # group's password
    login = des XQkR21zMB0TDU
}

user = fred {
    # fred is a member of group admin. He inherits the group's
password
    # as he does not have his one of his own.
    member = admin
}

group = admin {
    # group admin has a cleartext password which all members
share
    # unless they have their own password defined
    login = cleartext foobar
}

```

NOTE: The C program built from generate_passwd.c may be used to hand-generate encrypted passwords, or they may be taken from a Unix passwd file.

2). Authentication using passwd(5) files.

For selected users, you can perform DES authentication using existing passwd(5) files instead of entering the password into the configuration file directly (though using passwd(5) files is noticeably less efficient for large files).

You can specify this behavior per-user, by naming a passwd(5) file

in the password declaration (instead of giving a DES password), as follows:

```
user = fred {
    # look in file /etc/tac_plus_passwords to authenticate this
user    login = file /etc/tac_plus_passwords
}
```

3). Authentication using s/key.

If you have successfully built and linked in a suitable s/key library and compiled tac_plus to use s/key, you can then specify that a user be authenticated via s/key, as follows:

```
user = fred {
    login = skey
}
```

RECURSIVE PASSWORD LOOKUPS -----

As stated earlier, authentication passwords are looked up recursively:

The daemon looks first to see if the user has her own password. If not, it looks to see if she belongs to a group which has a password. This process recurses through the hierarchy of groups (a group can be a member of another group) until a password is found, or there are no more groups.

CONFIGURING DEFAULT AUTHENTICATION -----

By default, an unrecognized user will be denied authentication (NOTE: there is no way to authenticate someone with no username).

At the top level of the configuration file, you can set the default authentication to use a passwd(5) file, viz:

```
default authentication = file /etc/passwd
```

The effect of this statement is that if a user does not appear in the configuration file, the daemon will attempt to authenticate the user using passwords from this file i.e. /etc/passwd in this example.

If you have passwd(5) files from previous versions of tacacs daemons, this facility allows you to authenticate using the passwd(5) from older versions of tacacs, while you migrate to using the new configuration file.

CONFIGURING AUTHENTICATION ON THE NAS -----

On the NAS, configure authentication at login on all lines
(including
vty and console lines)

```
aaa new-model
aaa authentication login default tacacs+
```

NOTE: As soon as you issue this command, you will no longer be able
to
create new logins to your NAS without a functioning tacacs+ daemon
appropriately configured with usernames and password, so make sure
you
have this ready.

As a safety measure while setting up, we suggest you configure an
enable password and make it the last resort authentication method,
so
if your tacacs+ daemon fails to respond you will be able to use the
enable password to login. To do this, configure:

```
enable secret foo
aaa authentication login default tacacs+ enable
```

CONFIGURING AUTHORIZATION

Authorization must be configured on both the NAS and the daemon to operate correctly. By default, the NAS will allow everything until you configure it to make authorization requests to the daemon.

On the daemon, the opposite is true: The daemon will, by default, deny authorization of anything that isn't explicitly permitted.

Authorization allows the daemon to deny commands and services outright, or to modify commands and services on a per-user basis. Authorization on the daemon is divided into two separate parts: commands and services.

AUTHORIZING COMMANDS

Exec commands are those commands which are typed at a Cisco exec prompt. When authorization is requested by the NAS, the entire command is sent to the tac_plus daemon for authorization.

Command authorization is configured by specifying a list of egrep-style regular expressions to match command arguments (see the supplied man page, regexp.3, for a full description of regular expressions) and an action which is "deny" or "permit".

The following configuration example permits user Fred to run the following commands:

```
telnet 131.108.13.<any number> and
telnet 128.<any number>.12.3 and
show <anything>
```

All other commands are denied (by default).

```
user=fred {
    cmd = telnet {
        # permit specified telnets
        permit 131\.108\.13\.[0-9]+
        permit 128\.[0-9]+\.[12]\.3
    }
    cmd = show {
        # permit show commands
        permit .*
    }
}
```

NOTE: If an argument list you specify contains spaces or tabs, you must enclose it in double quotes.

The command and arguments which the user types gets matched to the

regular expressions you specify in the configuration file (in order of appearance). The first successful match performs the associated action (permit or deny). If there is no match, the command is denied by default.

Conversely, the following configuration example can be used to deny the command:

```
telnet 131.108.13.<any number>
```

and permit all other arguments, since the last line will match any argument list. All other commands and services are permitted due to the "default service = permit" clause.

Note: the default statement must be the first in the user clause

```
user=fred {
  default service = permit
  cmd = telnet {
    # allow all fred's telnet commands except to 131.108.13.*
    deny 131\.108\.13\.[0-9]+
    permit .*
  }
}
```

Note: Matches are not anchored, so "deny 131.108.13.[0-9]+" matches anywhere in the command. To anchor the match, use ^ at the beginning of the regular expression.

Note: When a command has multiple arguments, users may enter them in many different permutations. It can be cumbersome to create regular expressions which will reliably authorize commands under these conditions, so administrators may wish to consider other methods of performing authorization e.g. by configuring NAS-local privileged enable levels on the NAS itself.

CONFIGURING DEFAULT AUTHORIZATION

There are 3 places where defaults for authorization may be configured. Unless specified to the contrary, the default is always to deny authorization.

1). To override the default denial of authorization for a non-existent user, specify at the top level of the configuration file:

```
default authorization = permit
```

2). At the user level i.e. inside the braces of a user declaration, the default for a user who doesn't have a service or command explicitly authorized is to deny that service or command. The following directive will permit the service or command by default instead:

```
default service = permit
```

NOTE: This directive must appear first inside the user declaration.

3). At the service authorization level i.e. inside the braces of a service declaration, arguments in an authorization request are processed according to the algorithm described later. Some actions when authorizing services (e.g. when matching attributes are not found) depend on how the default is configured. The following declaration changes the default from deny to permit for this user and service.

```
default attribute = permit
```

NOTE: This directive must appear before any others inside the service declaration.

NOTE: for command authorization (as opposed to service authorization being discussed here), you specify deny .* or permit .* as the last line of the regular expression matches to create default behaviour.

AUTHORIZING EXEC STARTUP

If you authorize some exec commands, you implicitly agree to allow that user to start an exec (it doesn't make sense to permit exec commands if an exec can't be started to run those commands)

In addition to agreeing to allow an exec to start, you can supply some parameters whenever an exec starts e.g. an autocmd, a dialback string or a connection access list (acl).

In the example below, when an exec is started on the NAS, an acl of 4 will be returned to the NAS:

```
user=fred {  
  
    # this following line permits an exec to start and permits  
    # all commands and services by default  
  
    default service = permit  
  
    service = exec {  
        # When an exec is started, its connection access list will  
be 4.  
        # It also has an autocmd.  
        acl = 4  
        autocmd = "telnet foobar"  
    }  
  
    cmd = telnet {  
        # allow all fred's telnet commands except telnet to  
131.108.13.*  
        deny 131\.108\.13\.[0-9]+  
        permit .*    }  
}
```

```
}  
}
```

AUTHORIZING EXEC, SLIP, PPP and ARAP SERVICES

Authorizing exec, slip. PPP and arap services is different from command authorization.

When authorizing the above services, the NAS sends a request containing a number of attribute-value (AV) pairs, each having the form

attribute=value

(Note: during debugging, you may see AV pairs whose separator character is a "*" instead of a "=" sign. This is to signify whether a value in a pair is mandatory or optional. An "=" sign indicates a mandatory value. A "*" denotes an optional value).

e.g. a user starting ppp/ip using an address of 131.108.12.44 would generate a request with the following AV pairs:

addr=131.108.12.44
inacl=5

CONFIGURING SERVICE AUTHORIZATION

A similar list of AV pairs is placed in the daemon's configuration file in order to authenticate the service. The daemon compares each NAS AV pair to its configured AV pairs and either allows or denies the service. If the service is allowed, the daemon may add, change or delete some AV pairs before returning them to the NAS, thereby restricting what the user is permitted to do.

The complete algorithm by which the daemon processes its configured AV pairs against the list the NAS sends, is given below.

The Authorization Algorithm

For each AV pair sent from the NAS:

If the AV pair from the NAS is mandatory:

- a). look for an exact attribute,value match in the daemon's mandatory list. If found, add the AV pair to the output.
- b). If an exact match doesn't exist, look in the daemon's optional list for the first attribute match. If found, add the NAS AV pair to the output.
- c). If no attribute match exists, deny the command if the

default is to deny, or,

d). If the default is permit, add the NAS AV pair to the output.

If the AV pair from the NAS is optional:

e). look for an exact attribute,value match in the mandatory list. If found, add DAEMON's AV pair to output.

f). If not found, look for the first attribute match in the mandatory list. If found, add DAEMONS's AV pair to output.

g). If no mandatory match exists, look for an exact attribute,value pair match among the daemon's optional AV pairs. If found add the DAEMON's matching AV pair to the output.

h). If no exact match exists, locate the first attribute match among the daemon's optional AV pairs. If found add the DAEMON's matching AV pair to the output.

i). If no match is found, delete the AV pair if default is deny, or

j). If the default is permit add the NAS AV pair to the output.

k). After all AV pairs have been processed, for each mandatory DAEMON AV pair, if there is no attribute match already in the output list, add the AV pair (add only one AV pair for each mandatory attribute).

RECURSIVE AUTHORIZATION

Remember that authorization is also recursive over groups, in the same way that password lookups are recursive. Thus, if you place a user in a group, the daemon will look in the group for authorization parameters if it cannot find them in the user declaration.

EXAMPLES

key = "your key here"

```
user=fred {
  login = des mEX027bHtzTlQ
  name = "Fred Flintstone"
  member = administrators
  expires = "May 23 2005"
  arap = cleartext "Fred's arap secret"
  chap = cleartext "Fred's chap secret"

  service = exec {
```

```

    # When Fred starts an exec, his connection access list is 5
    acl = 5

    # We require this autocmd to be done at startup
    autocmd = "telnet foo"
}

# All commands except telnet 131.108.13.* are denied for Fred
cmd = telnet {

    # Fred can run the following telnet command

    permit 131\.108\.13\.[0-9]+
    deny .*
}

service = ppp protocol = ip {
    # Fred can run ip over ppp only if he uses one
    # of the following mandatory addresses If he supplies no
    # address, the first one here will be mandated

    addr=131.108.12.11
    addr=131.108.12.12
    addr=131.108.12.13
    addr=131.108.12.14

    # Fred's mandatory input access list number is 101
    inacl=101

    # We will suggest an output access list of 102, but Fred may
    # choose to ignore or override it

    optional outacl=102
}

service = slip {

    # Fred can run slip. When he does, he will have to use
    # these mandatory access lists

    inacl=101
    outacl=102
}

# set a timeout in the lcp layer of ppp
service = ppp protocol = lcp {
    timeout = 10
}
}

user = wilma {

    # Wilma has no password of her own, but she's a group member so
    # she'll use the group password if there is one. Same for her
    # password expiry date

    group = admin
}

```

```

group = admin {
    # group members who don't have their own password will be looked
    # up in /etc/passwd

    login = file /etc/passwd

    # group members who have no expiry date set will use this one

    expires = "Jan 1 1997"
}

```

USING PROGRAMS TO DO AUTHORIZATION

There are some limitations to the authorization that can be done using a configuration file. The main ones are that you're constrained by the algorithm the daemon uses, and that the configuration is basically static, so if you're trying to use it to allocate dynamic things (such as addresses from a pool) that varies over time, you need another mechanism.

The solution is to arrange for the daemon to call your own user-supplied programs to control authorization. These "callouts" permit almost complete control over authorization, allowing you to read all the fields in the authorization packet sent by the NAS including all its AV pairs, and to set authorization status and send a new set of AV pairs to the NAS in response.

USING AV PAIRS FOR AUTHORIZATION

During authorization, the NAS sends an authorization request packet containing various fields of interest and a set of AV pairs (see the tacacs+ protocol specification for a list of fields and pairs).

Fields from the authorization packet can be supplied to the programs you call on their command line, by using the appropriate dollar variables in the configuration file (see below).

AV pairs from the authorization packet are fed to the program's standard input, one per line. The program is expected to process the AV pairs and write them to its standard output, one per line. What happens then is determined by the exit status of the program.

NOTE: AV pairs are text strings with the format attribute=value. Unlike the configuration file which allows spaces when specifying AV pairs, there should be no spaces surrounding the "=" sign when using the programmatic interface.

CALLING SCRIPTS BEFORE AUTHORIZATION

You can specify a per-user program to be called before any other attempt to authorize is made by using a "before" clause e.g.

```
user = auth1 {
    before authorization "/usr/local/bin/pre_authorize $user $port
$address"
}
```

The AV pairs sent from the NAS will be supplied to this program's standard input, one pair per line.

Fields from the initiating authorization packet which the NAS sends to the daemon can also be passed to the program by using dollar variables in the command line. A complete list of available variables is as follows (consult the API specification for more details).

```
user      -- user name
name      -- Nas name
port      -- Nas port
address   -- Nac address (remote user location)
priv      -- privilege level (a digit, 0 to 15)
method    -- (a digit, 1 to 4)
type      -- (a digit, 1 to 4)
service   -- (a digit, 1 to 7)
status    -- (pass, fail, error, unknown)
```

Unrecognized variables will appear as the string "unknown".

If the program returns a status of 0, authorization is unconditionally permitted. No further processing is done on this request and no AV pairs are returned to the NAS.

If the program returns a status of 1, authorization is unconditionally denied. No further processing is done on this request and no AV pairs are returned to the NAS.

If the program returns a status of 2, authorization is permitted. The program is expected to modify the AV pairs that it receives on its standard input (or to create entirely new ones) and to write them, one per line, to its standard output. The new AV pairs will be sent to the NAS with a status of AUTHOR_STATUS_PASS_REPL. No further processing takes place on this request.

Any other status value returned from the program will cause an error to be returned to the NAS.

CALLING PROGRAMS AFTER AUTHORIZATION

You can specify a per-user program to be called after authorization processing has been carried out by the daemon (but before the authorization status and AV pairs have been transmitted to the NAS).

The program can optionally modify the AV pairs being sent back to the NAS and change the authorization status if required.

```
group = auth1 {
    # call /usr/local/bin/post_authorize passing it the username,
    port
    # and current authorization status.
    after authorization "/usr/local/bin/post_authorize $user $port
    $status"
}
```

The AV pairs resulting from the authorization algorithm that the daemon proposes to return to the NAS, are supplied to the program on standard input, one AV pair per line, so they can be modified if required.

Fields from the incoming authorization packet which the NAS sent to the daemon can also be passed to the program on its command line by specifying dollar variables in the command line (see previous section).

The program is expected to process the AV pairs and write them to its standard output, one per line. What happens then is determined by the exit status of the program:qq

If the program returns a status of 0, authorization continues as if the program had never been called. Use this if e.g. you just want a program to send mail when an authorization occurs, without otherwise affecting normal authorization.

If the program returns a status of 1, authorization is unconditionally denied. No AV pairs are returned to the NAS. No further authorization processing occurs on this request.

If the program returns a status of 2, authorization is permitted and any AV pairs returned from the program on its standard output are sent to the NAS in place of any AV pairs that the daemon may have constructed.

Any other value will cause an error to be returned the the NAS by the daemon.

WARNINGS

Pre and post authorization programs are invoked by handing the command

line to the Bourne shell. On many Unix systems, if the shell doesn't find the specified program it returns a status of one, which denies authorization. However, at least one Unix system (BSDI) returns a status code of 2 under these circumstances, which will permit authorization, and probably isn't what you intended.

Note also that if the program you call hangs, the authorization will time out and return an error on the NAS, and you'll tie up a process slot on the daemon host, eventually running out of resources. There is no special code to detect this in the daemon.

Unless you make special arrangements, the daemon will run as root and hence the programs it invokes will also run as root, which is a security weakness. It is strongly recommended that you use absolute pathnames when specifying programs to execute, and that you use the Makefile options TAC_PLUS_USERID and TAC_PLUS_GROUPID so that the daemon is not running as root when calling these programs,

The daemon communicates with pre and post authorization programs over a pair of pipes. Programs using the standard i/o library will use full buffering in these circumstances. This shouldn't be a problem for most programs, since they'll read AV pairs till they see end of file on input, and they'll flush all output when they exit.

CONFIGURING AUTHORIZATION ON THE NAS

If authorization is not explicitly configured on the NAS, no authorization takes place i.e. effectively, everything is permitted. Note that this is the converse of what happens on the daemon, where anything not explicitly permitted is denied by default.

To configure command authorization on the NAS, issue the following NAS configuration commands:

```
aaa authorization commands 1 tacacs+
aaa authorization commands 15 tacacs+
```

This will make the NAS send tacacs+ requests for all level 1 (ordinary user) and level 15 (privileged level) commands on all lines/interfaces.

NOTE: As soon as you configure the above on your NAS, you will only be permitted to execute NAS commands which are permitted by your tacacs+ daemon. So make sure you have configured, on the daemon, an authenticated user who is authorized to run commands, or you will be

unable to do much on the NAS after turning on authorization.

Alternatively, or in addition, you may also want to configure the following:

```
aaa authorization commands 1 tacacs+ if-authenticated
```

This will use tacacs+ authorization for level 1 (user-level commands) but if problems arise, you can just switch off the tacacs+ server and authorization will then be granted to anyone who is authenticated.

The following daemon configuration should be sufficient to ensure that you can always login as username "admin" (with a suitable password) and run any command as that user:

```
user = admin {  
    login = des kppPfHq/j6gXs  
    default service = permit  
}
```

ACCOUNTING -----

There is only one configurable accounting parameter -- the accounting file name. All accounting records are written, as text, to this filename compiled into the code. The filename is configured as follows at the top-level of the configuration file:

```
accounting file = <filename>
```

Since accounting requests occur (and are serviced) asynchronously, it is necessary to lock the accounting file so that two writers don't simultaneously update it. The code uses the fcntl call to do this locking. It is recommended that the accounting file reside on a local filesystem. Although fcntl locking over NFS is supported on some Unix implementations, it is notoriously unreliable. Even if your implementation is reliable, locking is likely to be extremely inefficient over NFS.

NAS CONFIGURATION -----

To get accounting records equivalent to previous versions of tacacs, the following is sufficient. "Stop" records contain elapsed time for connections and exec sessions.

```
aaa accounting network stop-only tacacs+  
aaa accounting exec stop-only tacacs+
```

DEBUGGING CONFIGURATION FILES

When creating configuration files, it is convenient to check their syntax using the `-P` flag to `tac_plus` e.g.

```
tac_plus -P -C <config file name>
```

will syntax check the configuration file and print any error messages on the terminal.

DEBUGGING A RUNNING SERVER

There is a myriad of debugging values that can be used in conjunction with the `-d` flag to produce debugging output in `/var/tmp/tac_plus.log`.

For example, starting the daemon with

```
tac_plus -C CONFIG -d 16
```

will put authentication debugging into `/var/tmp/tac_plus.log`. You can view this information by using the `tail` command.

```
tail -f /var/tmp/tac_plus.log
```

See the man page for more information.

CHANGING CONFIGURATIONS

To change a configuration file, you must edit the configuration file and then send the daemon a `SIGUSR1`. This will cause it to reinitialize itself and re-read the configuration file.

On startup, `tac_plus` creates the file `/etc/tac_plus.pid`, if possible, containing its process id, so something like the following should work:

```
# kill -USR1 `cat /etc/tac_plus.pid`
```

It's a good idea to check that the daemon is still running after sending it a `SIGUSR1`, since a syntactically incorrect configuration file will cause the daemon to die.

NOTE: The perl script `generate_passwd.pl` may be used to hand-generate encrypted passwords, or they may be taken from a Unix `passwd` file.

FREQUENTLY ASKED QUESTIONS

Q). I'm sure I configured it correctly, but accounting still doesn't work.

A). You will find that although you can configure accounting in 10.3, and it outputs debug messages, it doesn't send any accounting packets. This is because Accounting only works from 11.0 onwards.

Q). Does TACACS+ use a database instead of a flat (/etc/passwd like) file to decrease search times, say if we are talking about a large database of 40,000 users?

A). The TACACS+ authentication database is held internally as a hash table. This makes lookup times fast and fairly linear, at the expense of making the server use potentially large amounts of memory space. NOTE: If you specify that the server uses passwd(5) files for authentication, then you don't get this speed benefit, but you save space.

If you're willing to write the code, it should be a relatively simple matter to interface the code to a database scheme e.g. unix dbm files, or some proprietary database package, if you wish.

Q). Is there any way to avoid having clear text versions of the ARAP and CHAP secrets in the configuration file?

Both CHAP and ARAP require that the server knows the cleartext password (or equivalently, something from which the server can generate the cleartext password). Note that this is part of the definition of CHAP and ARAP, not just the whim of some Cisco engineer who drank too much coffee one morning.

If we encrypted the CHAP and ARAP passwords in the database, then we'd need to keep a key around so that the server can decrypt them when CHAP or ARAP needs them. So this only ends up being a slight obfuscation and not much more secure than the original scheme.

In regular TACACS, the CHAP and ARAP secrets were separated from the password file because the password file may be a system password file and hence world readable. But with TACACS+'s native database, there is no such requirement, so we think the best solution is to read-protect the files. Note that this is the same problem that a kerberos server has. If your security is compromised on the kerberos server, then your database is wide open. Kerberos does encrypt the database, but if you want your server to automatically restart, then you end up having to "kstash" the key in a file anyway and you're back to the same security problem.

So storing the cleartext password on the security server is really an absolute requirement of the CHAP and ARAP protocols, not something

imposed by TACACS+. All TACACS+ imposes is that you must send that cleartext password to the NAS in order to do CHAP or ARAP. While that may bother you, remember that the password is encrypted inside the TACACS+ packet.

We could have chosen a scheme where the NAS sends the challenge information to the TACACS+ daemon and the daemon uses the cleartext password to generate the response and returns that, but that means that we must include specific protocol knowledge into the protocol for both ARAP and CHAP and we would have to update the protocol every time a new authentication protocol is added. Hence we decided to go with the SENDPASS mechanism.

Q). How is the typical login authentication sequence done?

A). NAS sends START packet to daemon
 Daemon send GETUSER containing login prompt to NAS
 NAS prompts user for username
 NAS sends pkt to daemon
 Daemon sends GETPASS containing password prompt to the NAS
 NAS prompts user for password
 NAS sends pkt to daemon
 Daemon sends accept, reject or error to NAS

Q). Is there a GUI for the configuration file?

A). No. Use your favourite text editor.

Q). What does "default service = permit" really do?

A). When a request comes in to authorize exec startup, or ppp (with protocol lcp, ip, ipx), or slip, or arap or a specific command, the daemon looks for a matching declarations for the user (or groups the user is a member of).

For exec startup, it looks for a "service=exec" OR any command configured.

For ppp, it looks for a "service=ppp" and "protocol=(one of lcp, ip, ipx)".

For slip there must be a "service=slip" and for arap a "service=arap" clause.

For specific commands, there must be a matching cmd=<cmdname>.

If these aren't found, authorization will fail, *unless* you say "default service = permit".

Q). How do I make PAP work?

A). Avoid using PAP if possible. It's not very secure. If you must use

it, PAP logins are treated like ordinary user logins, with the username and password supplied by the remote host, so you need to declare a user in the Daemon configuration file which has a password, in order to process the PAP request.

Q). How can I deny some one from telneting from a commserver by ip address only. i.e. when command is 10.0.1.6 rather than telnet 10.0.1.6.

A). Configure "transport preferred none" on the lines in question. This will force the user to type "telnet 10.0.1.6" and this can be authorized in the normal fashion.

Q). I have an autocommand configured in the NAS-local database and I'm using "aaa authentication local-override". the autocommand doesn't work, but the username/password does. Why?

A). The "local-override" only applies to the authentication portion of the local database, so if you want an autocommand for this user, you need to also do:

```
aaa authorization exec local if-authenticated
```

This will use the local DB entry if one exists, allow authenticated users otherwise, or fail.

We don't have a "aaa authorization local-override" like we do for authentication. Unlike authentication, the local method for authorization is sort of equivalent to a local-override.

Q). Can tacacs+ only be enabled on a global basis? I want to selectively turn it on for, e.g. only modem-connected lines. How do I do this?

A). You turn tacacs+ ON on a global basis, but you can then change the behavior of individual lines to whatever you want, e.g.

```
aaa authentication login default tacacs+ none
aaa authentication login oldstyle line
aaa authentication login none none
line 1 16
login authentication default
line vty 0 4
login authentication oldstyle
line 0
login authentication none
```

Q). What are the memory recommendations for TACACS+?

A). Unless you're using passwd style files, TACACS+ holds lists in hash tables in memory. The overhead is modest e.g. each user entry occupies 64 bytes, plus space for strings like username etc. Access time is pretty constant regardless of number of users. On a sparc 2,

a
config file containing 2000 users requires about 0.5M of swap.

CANNED CONFIGURATIONS

Here are some canned configurations for getting demos started:

1). A canned configuration for login authentication only. This allows user fred to login with password "abcdef". If the tacacs+ server dies, the enable secret will be accepted as a login password instead.

DAEMON:

```
key = <some key>
```

```
# repeat as necessary for each user
user = fred {
    login = cleartext abcdef
}
```

NAS:

```
aaa new-model
enable secret foobar
! use tacacs+. If server dies, use the enable secret password
aaa authentication login default tacacs+ enable
tacacs-server host <some host ip address>
tacacs-server key <some key>
```

2). A canned configuration for command authorization. This will allow user fred to login with password abcdef and to run the privileged (level 15) commands 'write terminal' and 'configure'. All other privileged commands will be denied.

The "none" keyword in the NAS configuration line means that if the tacacs+ server dies, any command will be allowed.

DAEMON:

```
key = <some key>
```

```
# repeat as necessary for each user
user = fred {
    login = cleartext abcdef
    cmd = write {
        permit terminal
    }
    cmd = configure {
        permit .*
    }
}
```

NAS:

```
aaa new-model
! all level 15 (privileged commands). If server dies, allow
everything
aaa authorization commands 15 tacacs+ none
tacacs-server host <some host ip address>
tacacs-server key <some key>
```

3). Canned configuration for network access authorization. This config allows a fred to login to line 1 with password abcdef (or to and to run ppp using chap authentication. The chap password is "lab".

DAEMON:

```
key = <some key>

# repeat as necessary for each user
user = fred {
    login = cleartext abcdef
    chap = cleartext lab
    service = ppp protocol = ip {
        addr=1.0.0.2
    }
}
```

NAS:

```
aaa new-model
! authenticate exec logins (if not autoselecting)
aaa authentication login default tacacs+
! authorize network services via tacacs+
aaa authorization network tacacs+
! use tacacs+ for authenticating ppp users
aaa authentication ppp default tacacs+
tacacs-server host <some host ip address>
tacacs-server key <some key>
interface Async1
ip address 1.0.0.1 255.0.0.0
async default ip address 172.21.14.55
encapsulation ppp
async dynamic address
async mode interactive
! use chap to authenticate ppp users
ppp authentication chap
line 1
! need "modem inout" here and flow control if using a modem
```

CANONICAL LIST OF COMMANDS BY SCOPE

The following is a complete list of directives allowed in a tac_plus configuration file.

GLOBAL SCOPE (i.e. at the top-level of the configuration file):

```
default authentication = file <file name string>
```

```
accounting file = <filename string>
default authorization = permit
key = <string>
user = <username string> {}
group = <username string> {}
```

USER AND GROUP:

```
default service = permit
name = <string>
login = cleartext <password string> | des <des string> | skey | file
<file name string>
member = <group name string>
iacl = <number>
oacl = <number>
expires = <date string>
arap = cleartext <password string>
chap = cleartext <password string>
global = cleartext <password string>
before authorization = <program name string>
after authorization = <program name string>
service = <major protocol string> [protocol = <minor protocol
string>] {}
cmd = <command name string> {}
```

EXEC COMMAND:

```
permit <command arguments regexp string>
deny <command arguments regexp string>
```

SERVICE:

```
default attribute = permit
```

Authorization AV pairs

The following authorization AV pairs are supported by 10.3(3) and 11.0, except where noted.

The following AV pairs specify which service is being authorized. They

are typically accompanied by protocol AV pairs and other, additional pairs from the lists below.

```
service=arap
service=shell (for exec startup, and also for command
authorizations)
service=ppp
service=slip
```

```
service=system (not used).
```

```
protocol=lcp
```

The lower layer of PPP, always brought up before IP, IPX, etc.

is brought up.

protocol=ip

Used with service=ppp and service=slip to indicate which protocol layer is being authorized.

protocol=ipx

Used with service=ppp to indicate which protocol layer is being authorized.

protocol=atalk

with service=ppp or service=arap

protocol=vines

for vines over ppp.

protocol=unknown

for undefined/unsupported conditions.

cmd (service=shell)

Used with service=shell, if cmd=NULL this is an authorization request for starting an exec e.g. cmd=

If cmd is non-null, this is a command authorization request, It contains the name of the command being authorized e.g. cmd=telnet

cmd-arg (service=shell)

When doing command authorization, the name of the command is given by a cmd= AV pair and each command argument is represented by a cmd-arg AV pair e.g.

cmd-arg=archie.sura.net

NOTE: 'cmd-arg' should never appear in a configuration file. It is used internally by the daemon to construct a string which is then matched against the regular expressions which appear

in a cmd clause in the configuration file.

acl (service=arap, service=shell)

For arap, this pair contains an access-list number.

For exec authorization (i.e. when service=shell and cmd=NULL)

it contains an access-class number, e.g. acl=2

inacl (service=ppp, protocol=ip)

This AV pair contains an IP input access list number for slip or PPP/IP e.g. inacl=2

outacl (service=ppp, protocol=ip)

This AV pair contains an IP output access list number for
slip
or PPP/IP e.g. outacl=4

addr (service=slip, or service=ppp and protocol=ip)

Currently the IP address the remote host should have in a
slip
or PPP/IP when a connection is made e.g. addr=1.2.3.4

routing (service=slip, or service=ppp and protocol=ip)

Equivalent to the /routing flag in slip and ppp commands.
Can
have as its value the string "true" or "false".

timeout (service=arap)

Time until arap session disconnects (in minutes),
e.g. timeout=60

autocmd (service=shell, cmd=NULL)

During exec startup, this specifies an autocommand, like the
autocommand option to the username configuration command,
e.g. autocmd="telnet foo.com"

noescape (service=shell, cmd=NULL)

During exec startup, this specifies "noescape", like the
noescape option to the username configuration command. Can
have as its value the string "true" or "false",
e.g. noescape=true

nohangup (service=shell, cmd=NULL)

During exec startup, this specifies "nohangup", like the
nohangup option to the username configuration command. Can
have as its value the string "true" or "false",
e.g. nohangup=true

priv-lvl (service=shell, cmd=NULL)

Specifies the current privilege level for command
authorizations, a number from zero to 15 e.g. priv_lvl=5.

Note: in 10.3 this attribute was priv_lvl i.e.
it contained an underscore instead of a hyphen.

zonelist (service=arap)

An Appletalk zonelist for arap equivalent to the line
configuration command "arap zonelist" e.g. zonelist=5

addr-pool (not in 10.3, service=ppp, protocol=ip)

to This AV pair specifies the name of a local pool from which
 get the address of the remote host.