TACACS+

Marc Huber

COLLABORATORS			
	TITLE : TACACS+		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Marc Huber	December 23, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction

tac_plus is a TACACS+ daemon. It provides Cisco Systems routers and access servers with authentication, authorisation and accounting services.

This version is a major rewrite of the original Cisco source code. However, it's outdated and unsupported. Feel encouraged to use tac_plus-ng.

Key features include:

- NAS specific host keys, prompts, enable passwords
- NAS- and ACL-dependent group memberships
- Flexible external backends for user profiles (e.g. via PERL scripts or C; LDAP (including ActiveDirectory), RADIUS and others are included)
- Connection multiplexing (multiple concurrent NAS clients per process)
- Session multiplexing (multiple concurrent sessions per connection, *single-connection*)
- Scalable, no limit on users, clients or servers.
- CLI context aware. At the time of writing this, no other TACACS+ daemon is.
- Both IPv4 and IPv6 are fully supported.
- Implements and autodetects HAProxy protocol 2.
- Compliant to the original TACACS+ protocol specification (draft 1.78).

1.1 Download

You can download the source code from the GitHub repository at https://github.com/MarcJHuber/event-driven-servers/. Documentation is available on the original site, https://www.pro-bono-publico.de/projects/, too.

2 Definitions and Terms

The following chapters utilize a couple of terms that may need further explanation:

NAC	A Network Access Client, e.g. the source host of a `telnet` connection.
NAS	A Network Access Server, e.g. a Cisco box, or any other client which makes TACACS+ authentication and authorization requests, or generates TACACS+ accounting packets.
Daemon	A program which services network requests for authentication and authorization, verifies identities, grants or denies authorizations, and logs accounting records.
AV pairs	Strings of text in the form `attribute=value`, sent between a NAS and a TACACS+ daemon as part of the TACACS+ protocol.

Since a *NAS* is sometimes 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*.

2.1 TACACS, XTACACS and TACACS+

As a tidbit of historical value, there are about three versions of authentication protocol that people may 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. Both are UDP based services.

The third one, and that's the version virtually everybody uses right now, is *TACACS+* or *tac_plus*, which is what is documented here. *TACACS+* is based on TCP and as such **not compatible** with any previous versions of *TACACS*.

Apparently, at least one vendor did implement *TACACS+*, but refers to it as *HWTACACS*. Chances are that there are no technical (but possibly marketing or legal) reasons behind that. Documentation is scarce.

3 Operation

This section gives a brief and basic overview how to run **tac_plus**.

In earlier versions, **tac_plus** wasn't a standalone program but had to be invoked by **spawnd**. This has changed, as **spawnd** functionality is now part of the **tac_plus** binary. However, using a dedicated **spawnd** process is still possible and, more importantly, the **spawnd** configuration options and documentation remain valid.

tac_plus may use auxiliary **MAVIS** backend modules for authentication and authorization.

3.1 Command line syntax

The only mandatory argument is the path to the configuration file:

```
tac_plus [ -P ] [ -d level ] [ -i child_id ] configuration-file [ id ]
```

If the program was compiled with CURL support, *configuration-file* may be an URL.

Keep the `-P` option in mind - it is imperative that the configuration file supplied is syntactically correct, as the daemon won't start if there are any parsing errors at start-up.

The `-d` switch enables debugging. You most likely don't want to use this. Read the source if you need to.

The `-i` option is only honoured if the build-in **spawnd** functionality is used. In that case, it selects the configuration ID for **tac_plus**, while the optional last argument *id* sets the ID of the **spawnd** configuration section.

3.2 Signals

Both the master (that's the process running the **spawnd** code) and the child processes (running the **tac_plus** code) intercept the `SIGHUP` signal:

- The master process will restart upon reception of `SIGHUP`, re-reading the configuration file. The child processes will recognize that the master process is no longer available. It will continue to serve the existing connections and terminate when idle.
- If `SIGHUP` is sent to a child process it will stop accepting new connections from its master process. It will continue to serve the existing connections and terminate when idle.

Sending `SIGUSR1` to the master process will cause it to abandon existing child processes (these will continue to serve the existing connections only) and start new child processes.

3.3 Event mechanism selection

Several level-triggered event mechanisms are supported. By default, the one best suited for your operating system will be used. However, you may set the environment variable `IO_POLL_MECHANISM` to select a specific one.

The following event mechanisms are supported (in order of preference):

- port (Sun Solaris 10 and higher only, `IO_POLL_MECHANISM=32`)
- kqueue (*BSD and Darwin only, `IO_POLL_MECHANISM=1`)
- /dev/poll (Sun Solaris only, `IO_POLL_MECHANISM=2`)
- epoll (Linux only, `IO_POLL_MECHANISM=4`)
- poll (`IO_POLL_MECHANISM=8`)
- select (`IO_POLL_MECHANISM=16`)

Environment variables can be set in the configuration file at top-level:

```
setenv IO_POLL_MECHANISM = 4
```

3.4 Sample Configuration

A single configuration file is sufficient for configuring **spawnd**, **tac_plus** and the *MAVIS* authentication and authorization backends.

The following **spawnd** configuration stanza accepts connections on TCP ports 49 and 4949 and forwards these to one of the **tac_plus** processes. The **tac_plus** configuration configures a couple of user groups, has one single user defined and relies on the *MAVIS* backend for additional users.

```
#!/usr/local/sbin/tac_plus
id = spawnd {
    listen = { port = 49 }
    listen = { port = 4949 }
    #
    # See the spawnd configuration guide for further configuration options.
}

id = tac_plus {
    accounting log = /var/log/tac_plus/%Y/%m/%d.log
    retire limit = 1000
    mavis module = external {
        exec = /usr/local/lib/mavis_tacplus_passwd.pl
        # see the MAVIS configuration manual for more options and other modules
    }
    login backend = mavis

    host = world {
        welcome banner = "\nHitherto shalt thou come, but no further. (Job 38.11)\n\n"
        key = QaWsEdRfTgY
        enable 15 = clear test
        address = 0.0.0.0/0
    }

    group = readwrite {
        default service = permit
        service = shell {
            default command = permit
            set priv-lvl = 15
```

```
        }
    }

    group = getconfig {
        default service = permit
        service = shell {
            set autocmd = "sho run"
            set priv-lvl = 15
        }
    }

    user = marc {
       login = crypt $1$xxxxxxxx$hDZPHghXe8XvoHeFdqUwm/
       member = readwrite@world
       member = getconfig@192.168.0.0/16
    }
}
```

3.5 Sample Configuration with Realms

Here's a more sophisticated example, acknowledging that most folks prefer to look at sample configurations instead of actually reading documentation.

```
syslog default = deny

id = spawnd {
  # Really, have a look at the spawnd configuration. There are
  # lots of useful options.

  listen = {
    port = 4949
  }

  listen = {
    port = 4950 realm = core
  }
}

id = tac_plus {
  client realm = adminRealm   # Could specify these
  aaa realm = local           # at host level, too.
  dns reverse-lookup = no

  realm = adminRealm {
    host = myPCs {
      address = 192.0.2.8,192.0.2.24/29
    }
  }

  realm = customerRealm {
    dns reverse-lookup = yes
    host = customer {
      address = 172.16.0.0/12
    }
  }

  # This will be placed in the "default" realm
  host = cpeRouters {
    address = 0.0.0.0/0
    key = myKey
```

```
  welcome banner = "Welcome C=%%C Time=%c\n"
  client realm = customerRealm
  aaa realm = ldapRealm
}

# And this one in the "core" realm. Could write this as
# realm = ... { host = ... }, too.
host realm core = coreRouters {
  address = 0.0.0.0/0
  key = CoreKey
  client realm = adminRealm
  aaa realm = unixRealm
}

realm = unixRealm {
  mavis module = groups {
    resolve gids = yes
    groups filter = /^(guest|staff)$/
    script out = {
      # copy the already filtered UNIX group access list to TACMEMBER
      eval $GIDS =~ /^(.*)$/
      set $TACMEMBER = $1
    }
  }

  mavis module = external {
    exec = /usr/local/sbin/pammavis "pammavis" "-s" "mavis"
  }

  user backend = mavis
  login backend = mavis chpass

  group = staff {
    service = shell {
      set priv-lvl = 15
      script = {
        if (cmd == "") permit # shell startup
        if (cmd =~ /^configure/) permit
      }
      message debug = "author '%c %a'"
      cmd = show { permit .* }
    }
  }

  group = guest {
    member = staff
    service = shell {
      cmd = show { permit .* }
      message deny = "denied by t+"
    }
  }
}

realm = ldapRealm {
  mavis module = external {
    exec = /usr/local/lib/mavis/mavis_tacplus_ads.pl
    setenv USE_TLS = 0
    setenv LDAP_HOSTS = "192.0.2.193"
    setenv LDAP_SCOPE = sub
    setenv LDAP_BASE = "dc=example,dc=com"
  }
  user backend = mavis
```

```
    login backend = mavis
    pap backend = mavis
  }

  realm = localRealm {
    user = marc {
      password = clear marc
      service = shell {
        default command = permit
      }
    }
  }
}
```

4 Configuration Syntax

The configuration stanzas covered by the following sections are to be placed between

```
id = tac_plus = {
```

and the corresponding

```
}
```

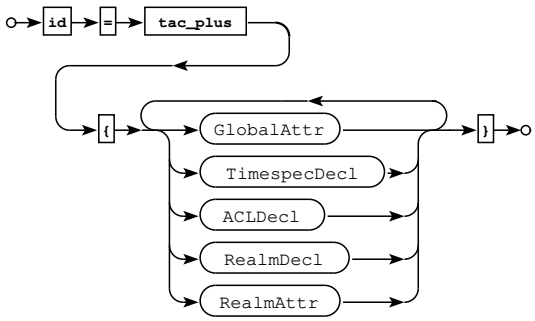
Comments in Configuration Files
Comments can appear anywhere in the configuration file, starting with the # character and extending to the end of the current line. Should you need to disable this special meaning of the # character, e.g. if you have a password containing a # character, simply enclose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

Including Files
Configuration files may refer to other configuration files:
`include =` *file*
will read and parse *file*. Shell wildcard patterns are expanded by `glob`*(3)*. The `include` statement will be accepted virtually everywhere (but not in comments or textual strings).

The recommended order for writing a configuration file is

1. global definitions
2. hosts
3. timespecs
4. acls
5. groups
6. users

The reasoning behind that non-random order is that parts of the configuration may use other parts, and these need to exist before being used.



Railroad diagram: TacPlusConfig

4.1 Global options

The global configuration section may contain the following configuration directives, plus the *realm* options detailed in the next section. *realm* options defined at global level will implicitely be assigned to the *default* realm and will be inherited to subsequently defined realms.

4.1.1 Limits and timeouts

A number of global limits and timeouts may be specified exclusively at global level:

- `retire limit =` *n*

  The particular daemon instance will terminate after processing *n* requests. The **spawnd** instance will spawn a new instance if necessary.

  Default: unset

- `retire timeout =` *s*

  The particular daemon instance will terminate after *s* seconds. **spawnd** will spawn a new instance if necessary.

  Default: unset

Time units
Appending `s`, `m`, `h` or `d` to any timeout value will scale the value as expected.

4.1.2 DNS

tac_plus can make use of static and, if compiled with c-ares support, of dynamic DNS entries. The relevant global configuration options at global level are:

- `dns cleanup period =` *s*

  Remove unused dynamic DNS data from cache after *s* seconds. (Default: 8 hours.)

- `dns preload address` *address* = *hostname*

  Preload DNS cache with *address*-to-*hostname* mapping.

- `dns preload file =` *filename*

  Preload DNS cache with *address*-to-*hostname* mappings from *filename* (see your `hosts`(5) manpage for syntax).

  Example:

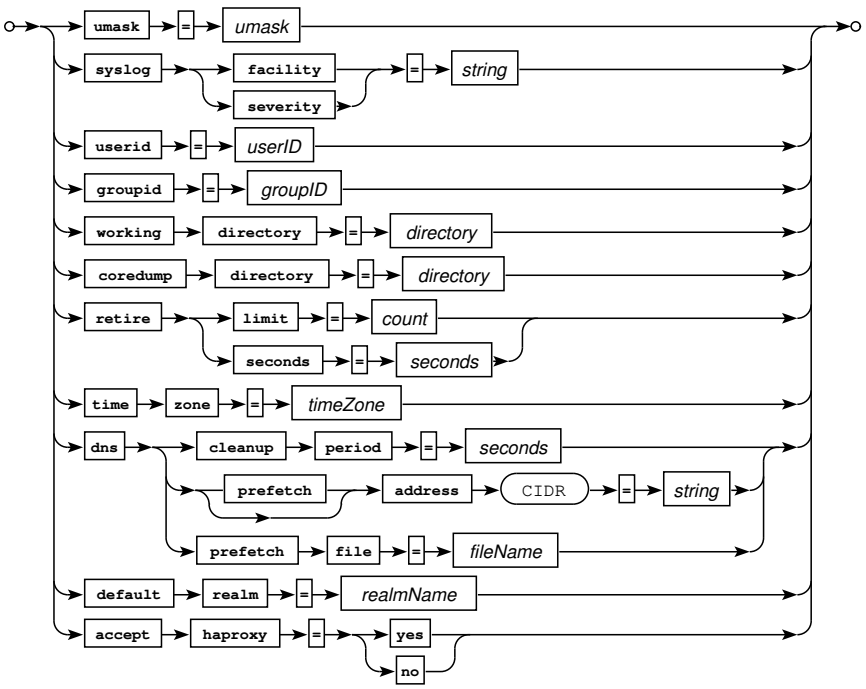
```
dns preload address 1.2.3.4 = router.example.com
dns preload file = /etc/hosts

host = router.example.com {
    # "address = 1.2.3.4" is actually implied
    key = mykey
}
```

4.1.3 Process-specific options

There are a couple of process-specific options available:

- `userid =` *uid*
  Run daemon under user id *uid*. This option is deprecated.
- `groupid =` *gid*
  Run daemon under group id *gid*. This option is deprecated.
- `working directory =` *directory*
  Changes the current working directory to *directory*.
- `coredump directory =` *directory*
  Dump cores to *directory*. You really shouldn't need this.
- `accept haproxy =(yes|no)`
  Enables HAProxy support. Disabled by default.

4.1.4 Railroad Diagrams



Railroad diagram: GlobalDecl

4.2 Realms

Historically, *realms* were introduced in listen directives in the **spawnd** configuration section:

```
spawnd = {
  listen = { port = 49 }
  listen = { port = 3939 }
  listen = { port = 4949 realm = oneRealm}
  listen = { port = 5959 realm = otherRealm}
}
```

Default Realm
If `spawnd` doesn't supply a realm name, then *tac_plus* will fall back to using the realm named `default`, which contains all the realm specific directives not defined in realm context.
You can select the actual default realm with

```
default realm = RealmName
```

at global configuration level.

The main task of the **spawnd** component is to accept a connection and forward it, including the associated realm, to the **tac_plus** process. **tac_plus** then uses that realm to assign the correct host object hierarchy to the connection.

The prefered syntax to use (and define) realms is

```
realm = realmName { ... }
```

at global configuration level. Realms may include hosts, users, groups, *MAVIS* configurations and various other configuration options. Realms and hosts may refer to other realms for authentication.

More precisely, the three uses for realms are:

- Authentication, Authorization and Accounting Realms

  Everything related to users and user groups, including logging, is configured in an AAA realm. The AAA realm to use may be selected at global, realm and host level, using the

  ```
  aaa realm = realmName
  ```

  directive.

- Network Access Server Realms

  This is the realm where the NAS host object was defined. Hosts and NAS specific parameters are defined in this realms context.

- Network Acccess Client Realms

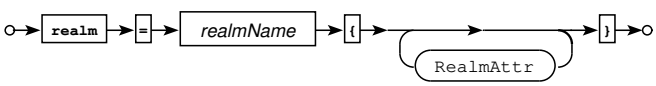
  NAC specific parameters may be retrieved from a NAC realm, which may be spefified at global, realm and host level:

  ```
  nac realm = realmName
  ```

  This may for example be used to enable DNS reverse lookups for remote clients on a per-realm basis.

Defining realms is optional. As mentioned before, realm options defined at top-level will be assigned to the `default` realm, which is actually the default for both the `aaa` and `nac` realms.

4.2.1 Railroad Diagrams



Railroad diagram: RealmDecl

4.3 Realm options

The following options may be specified at *realm* and *global* level:

4.3.1 Logging

The software provides logs for

- Authentication

```
  authentication log = log_destination
```

- Authorization

```
  authorization log = log_destination
```

- Accounting

```
  accounting log = log_destination
```

Logs may be written to multiple destinations:

```
  authentication log = log_1
  authentication log = log_2
...
  authentication log = log_n
```

Valid log destinations are:

- Files

  For logging to plain disk files `fcntl(2)` file locking is used, so it is recommended that file resides on a local filesystem. Although `fcntl` locking over NFS is supported on some Unix implementations, it is notoriously unreliable, and eVen if your implementation is reliable, locking is likely to be extremely inefficient over NFS.

  If the underlying file system supports atomic appends, `fcntl(2)` file locking may not be necessary. Prepending the log destination with a > character will turn file locking off and switch log file handling to synchronous mode.

```
  # async
  authentication log = /var/log/tac_plus/access1.log
  # sync:
  authentication log = ">/var/log/tac_plus/access2.log"
```

- Commands

  If the log destination starts with a `|` character a shell running the remainder of the destination argument will be invoked, and log messages will be feeded to that shell on standard input. Example:

```
  authentication log = "|exec /usr/bin/logger"
```

  (The `exec` isn't strictly necessary here, but avoids keeping an otherwise idle shell process around.)

- Syslog

  Logging to `syslogd(8)` can be enabled by either using the `syslog` keyword

```
  accounting log = syslog
```

  (which will use your systems `syslog(3)` function for logging, or by utilizing

```
  accounting log = Address[:UDPPort]
```

syntax.

Moreover, named log destinations may be used, e.g.:

```
log = mylog {
  destination = 169.254.0.23 # UDP syslog
  # or one of the following:
  # destination = [fe80::123:4567:89ab:cdef]:514 # IPv6 UDP, with non-standard UDP port
  # destination = "/tmp/x.log" # plain file
  # destination = ">/tmp/x.log" # plain file
  # destination = "|my_script.sh" # script
  # destination = syslog # syslog(3)
  #
  syslog compliance = RFC3164 # this is the default for UDP syslog
  # syslog compliance = RFC5424 # this loosely matches the new syslog standard
  syslog facility = MAIL # sets log facility
  syslog severity = DEBUG # sets log severity
  log separator = "|" # The actual default for syslog, for others it's "\t"
}
authentication log = mylog
accounting log = mylog
authorization log = mylog
```

Syslog
Logging to `syslogd`*(8)* can be disabled using

```
syslog default = deny
```

Log destinations may contain `strftime(3)`-style character sequences, e.g.:

```
authorization log = /var/log/tac_plus/%Y/%m/%d.auth
```

to automate time-based log file switching. By default, the daemon will use your local time zone for time conversion. You can switch to a different one by using the `time zone` option (see below).

There are a couple of other configuration options that may be useful:

- `date format =` *string*

  This defines a format string for date (and time) representation in subsequentially defined log files. Default:

  ```
  date format = "%Y-%m-%d %H:%M:%S %z"
  ```

- `log separator =` *string*

  This defines the CSV separator string for log entries in subsequentially defined log files. Default:

  ```
  log separator = "\t"
  ```

- `time zone =` *time-zone*

  By default, the daemon uses your local system time zone to convert the internal system time to calendar time. This option sets the `TZ` environment variable to the *time-zone* argument. See your local `tzset` man page for details.

- `umask =` *mode*

  This sets the file creation mode mask. Example:

  ```
  umask = 0640
  ```

- `authorization log group = (yes|no)`

  Set this to have the name of the last matching group appended to the username in authorization logs (separated by a single '/' character).

4.3.1.1 Accounting

All accounting records are written, as text, to the file (or command) specified with the `accounting log` directive.

Accounting records are text lines containing tab-separated fields. The first 6 fields are always the same. These are:

- timestamp
- NAS address
- username
- port
- NAC address
- record type

Following these, a variable number of fields are written, depending on the accounting record type. All are of the form `attribute=valu`
There will always be a `task_id` field.

Current attributes are:

```
unknown service start_time port elapsed_time status priv_level cmd protocol cmd-arg bytes_
bytes_out paks_in paks_out address task_id callback-dialstring nocallback-verify callback-
callback-rotary
```

More may be added over time.

Example records (lines wrapped for legibility) are thus:

```
1995-07-13 13:35:28 -0500  172.16.1.4  chein  tty5   198.51.100.141
        stop   task_id=12028  service=exec  port=5   elapsed_time=875
1995-07-13 13:37:04 -0500  172.16.1.4  lol    tty18  198.51.100.129
        stop   task_id=11613  service=exec  port=18  elapsed_time=909
1995-07-13 14:09:02 -0500  172.16.1.4  billw  tty18  198.51.100.152
        start  task_id=17150  service=exec  port=18
1995-07-13 14:09:02 -0500  172.16.1.4  billw  tty18  198.51.100.152
        start  task_id=17150  service=exec  port=18
```

Elapsed time is in seconds, and is the field most people are usually interested in.

On the NAS, 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 system default start-stop group tacacs+
aaa accounting exec default start-stop group tacacs+
aaa accounting exec acctlist stop-only group tacacs+
aaa accounting commands 15 acctlist start-stop group tacacs+
aaa accounting network acctlist stop-only group tacacs+

line XX
  accounting commands 15 acctlist
```

4.3.1.2 Spoofing Syslog Packets

The script `tacspooflog.pl` (which comes bundled with the distribution) may be used to make `syslogd` believe that logs come straight from your router, not from `tac_plus`.

E.g., if your `syslogd` is listening on `127.0.0.1`, you may try:

```
  access log = "|exec sudo /path/to/tacspooflog.pl 127.0.0.1"
```

This may be useful if you want to keep logs in a common place.

Please note that this will work for IPv4 only.

4.3.2 Limits and timeouts

A number of global limits and timeouts may be specified at realm and global level:

- `connection timeout =` *s*
  Terminate a connection to a NAS after an idle period of at least *s* seconds.
  Default: 600
- `context timeout =` *s*
  Clears context cache entries after *s* seconds of inactivity. Default: 3600 seconds.
  Default: 3600
- `warning period =` *d*
  Set warning period for password expiry to *d* days.
  Default: 14

4.3.2.1 DNS

tac_plus can make use of static and, if compiled with `c-ares` support, of dynamic DNS entries. The relevant realm configuration options are:

- `dns reverse-lookup =(yes|no)`
  This sets the global default for DNS reverse-lookups (Default: `no`).
- `dns timeout =` *Seconds*
  This directive specifies the maximum time to wait when doing DNS reverse lookups. (Default: `0`).

4.3.2.2 Authentication

- `password(max-attempts =` *integer* `|backoff =` *seconds* `|acl =` *acl* `)`
  `backoff` sets a backoff time for failed authentications. The daemon will wait for *seconds* seconds before returning a final authentication failure (password incorrect) message.
  The `max-attempts` parameter limits the number of `Password:` prompts per TACACS+ session at login. It currently defaults to `1`, meaning that a typical login sequence with bad passwords would look like:

```
> telnet 10.0.0.2
Trying 10.0.0.2...
Connected to 10.0.0.2.
Escape character is '^]'.

Welcome. Authorized Use Only.

Username: admin
Password: ***
Password incorrect.


Welcome. Authorized Use Only.

Username: admin
Password: ****
Password incorrect.


Welcome. Authorized Use Only.
```

```
Username: admin
Password: *
Password incorrect.

Connection closed by foreign host.
```

Using, for example,

```
password max-attempts = 3
```

(the actual default in earlier versions was `4`) would change this dialog to:

```
> telnet 10.0.0.2
Trying 10.0.0.2...
Connected to 10.0.0.2.
Escape character is '^]'.

Welcome. Authorized Use Only.

Username: admin
Password: ***

Password incorrect.
Password: ****

Password incorrect.
Password: *****

Password incorrect. Go away.


Welcome. Authorized Use Only.

Username:
```

It's at the NAS's discretion to restart the authentication dialog with a new TACACS+ session or to close the (Telnet/SSH/...) session to the user if TACACS+ authentication fails.

`password acl` may be used to perform simple compliance checks on user passwords. For example, to enforce a minimum password length of 6 characters you may try

```
acl script = password-compliance {
    if (password =~ /^....../)
        permit
    deny
}
password acl = password-compliance
```

Authentications using passwords that fail the check will be rejected.

- `anonymous-enable = ( permit | deny )`

  Several broken *TACACS+* implementations send no or an invalid username in `enable` packets. Setting this option to `deny` tries to enforce user authentication before enabling. This option defaults to `permit`.

  Alas, this may or may not work. In theory, the enable dialog should look somewhat like:

```
Router> enable
Username: me
Password: *******
Enable Password: **********
Router#
```

  However, some implementations may resend the user password at the `Enable Password:` prompt. In that case you've got only two options: Either use

```
    enable = login
```

at user group level, which will omit the secondary password query and let the user enable with his login password, or permit anonymous `enable` (which is disabled by default) with

```
    anonymous-enable = permit
```

(globally, or at host level) and use enable passwords defined at host level.

- `augmented-enable = ( permit | deny )`

  For *TACACS+* client implementations that send `$enable$` instead of the real username in an enable request, this will permit user specific authentication using a concatenation of username and login password, separated with a single space character:

```
> enable
Password: myusername mypassword
#
```

  `enable [ level ] = login` needs to be set in the users' profile for this option to take effect.

  Default: `augmented-enable = deny`

  `augmented-enable` will only take effect if the NAS tries to authenticate a username matching the regex

```
^\$enab..\$$
```

  (e.g.: `$enable$`, `$enab15$`). That matching criteria may be changed using an ACL:

```
acl script = custom_enable_acl { if (user =~ ^demo$) permit deny }
enable user acl = custom_enable_acl
```

4.3.2.3 Miscellaneanous

- `single-connection [ may-close ] = ( yes | no )`

  This directive may be used to permit or deny the single-connection feature. The `may-close` keyword tells the daemon to close a connection if it's unused (default: disabled). This directive may be overridden at host object level.

- `skip conflicting groups = ( yes | no )`

  If this is set, group conflicts will be ignored in `member` directives. The first group defined for a given NAS wins.

- `skip missing groups = ( yes | no )`

  If this is set, non-existing groups will be ignored in `member` directives.

- `client realm =` *realm*

  Lookup NAC parameters in realm *realm* instead of using the `default` one.

4.3.2.4 User backend options

These options are relevant for configuring the MAVIS user backend:

- `aaa realm =` *realmName*

  Use the aaa configuration (users, groups, mavis, ...) from realm *realm* instead of the `default` one.

- `group realm =` *realmName*

  Use the group configuration from realm *realm* instead of the `default` (or `aaa`) realm one.

- `pap password [ default ] = ( login | pap )`

  When set to `login`, the PAP password default for new users will be set to use the login password.

- `pap password mapping = ( login | pap )`

  When set to `login`, PAP authentication requests will be mapped to ASCII Login requests. You may wish to uses this for NEXUS devices.

  May be overridden at host level.

- `user backend = mavis`

  Get user data from the MAVIS backend. Without that directive, only locally defined users will be available and the MAVIS backend may be used for authenticating known users (with `password = mavis` or simlar) only.

- `pap backend = mavis [ prefetch ]`

  Verify PAP passwords using the MAVIS backend. This needs to be set to either `mavis` or `prefetch` in order to authenticate PAP requests using the MAVIS backend. If unset, the PAP password from the users' profile will be used.

  If `prefetch` is specified, the daemon will first retrieve the users' profile from the backend and then authenticate the user based on information eventually found there.

  This directive implies `user backend = mavis`.

- `login backend = mavis [ prefetch ] [ chalresp [ noecho ] ] [ chpass ]`

  Verify Login passwords using the MAVIS backend. This needs to be set to either `mavis` or `prefetch` in order to authenticate login requests using the MAVIS backend. If unset, the login password from the users' profile will be used.

  If `prefetch` is specified, the daemon will first retrieve the users' profile from the backend and then authenticate the user based on information eventually found there.

  This directive implies `user backend = mavis`.

  For use with OPIE-enabled MAVIS modules, add the `chalresp` keyword (and, optionally, add `noecho`, unless you want the typed-in response to display on the screen). Example:

```
login backend = mavis chalresp noecho
```

  For non-local users, if the `chpass` attribute is set and the user provides an empty password at login, the user is given the option to change his password. This requires appropriate support in the MAVIS backend modules.

- `mavis module =` *module* `{ ... }`

  Load MAVIS module *module*. See the MAVIS documentation for configuration guidance.

- `mavis path =` *path*

  Add *path* to the search-path for MAVIS modules.

- `mavis cache timeout =` *s*

  Cache MAVIS authentication data for *s* seconds. If *s* is set to a value smaller than `11`, the dynamic user object is valid for the current TACACS+ session only. Default is `120` seconds.

- `mavis noauthcache`

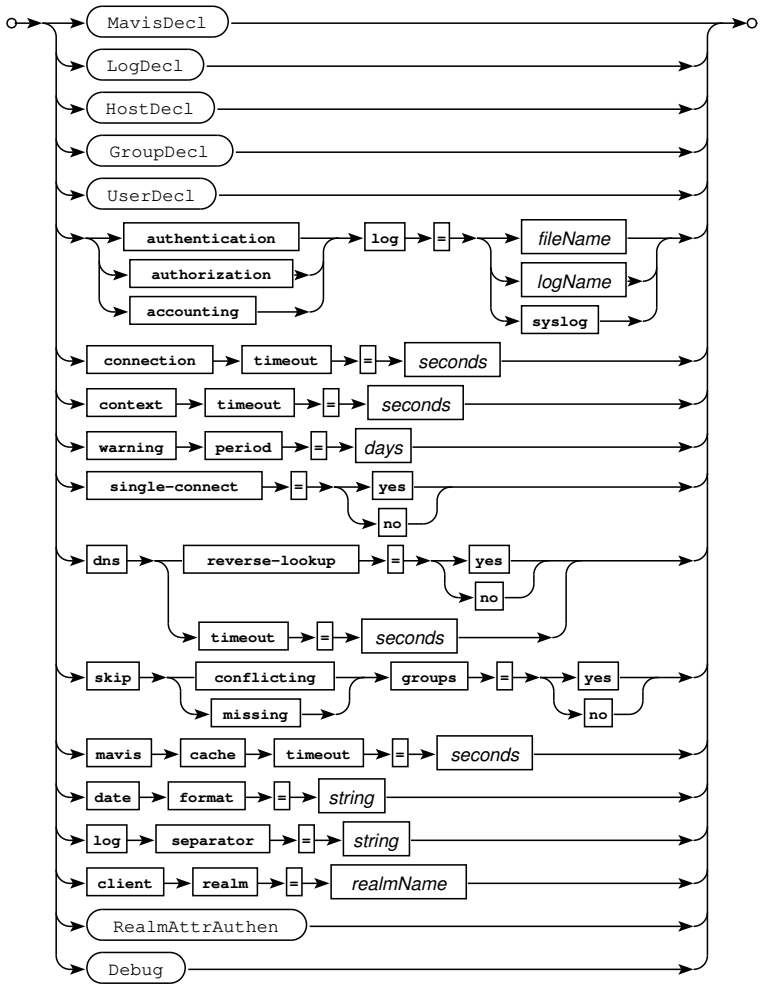
  Disables password caching for MAVIS modules.

- `mavis user filter = [ not ]` *acl*

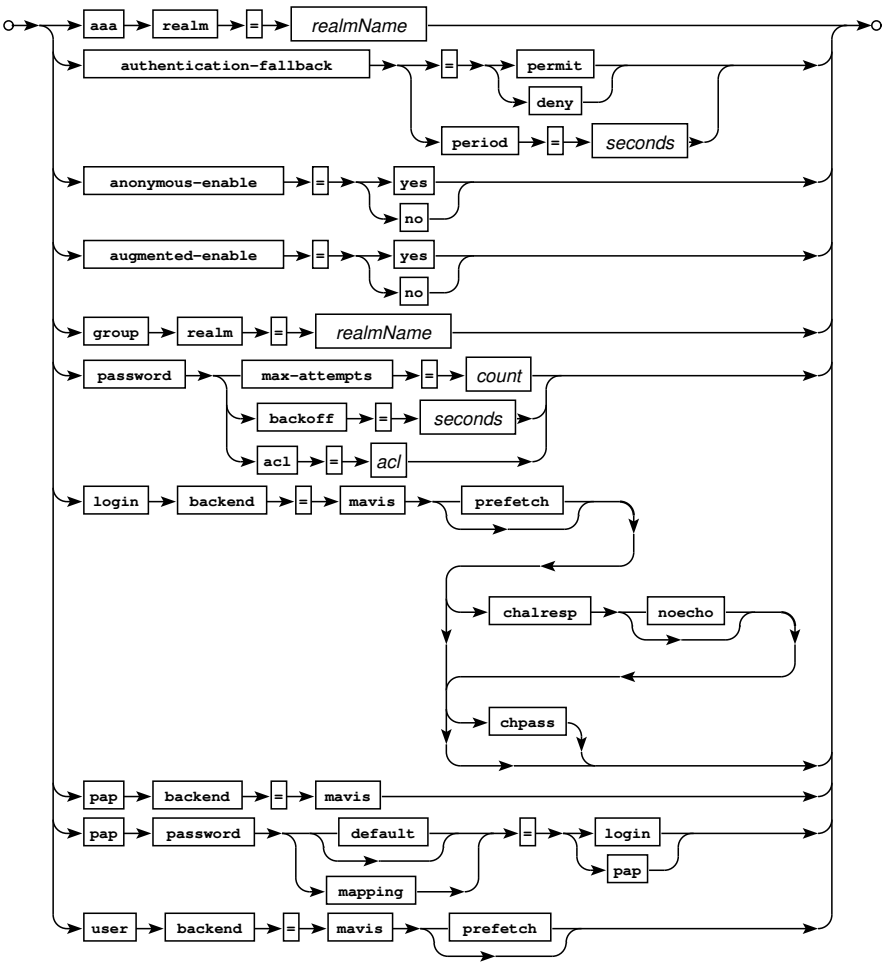
  Query MAVIS user backend only if *acl* matches. Defaults to:

```
acl script = __internal__username_acl__ { if (user =~ "[]<>/()|=[]+") deny permit }
mavis user filter = __internal__username_acl__
```

4.3.2.5 Railroad Diagrams



Railroad diagram: RealmAttr



Railroad diagram: RealmAttrAuthen

4.3.3 Hosts

The daemon will talk to known NAS addresses only. Connections from unknown addresses will be rejected.

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 there), then you'll have to specify an (non-empty) encryption key. The identical key must also be configured on any NAS which communicates with tac_plus.

To specify a global key, use a statement similar to

```
host = 0.0.0.0/0 {
  key = "your key here"
}
```

or, alternatively,

```
host = world {
  key = "your key here"
  address = 0.0.0.0/0
}
```

(where `world` is *not* a keyword, but just some arbitrary character string).

Double Quotes
You only need 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 key on the NAS.
The daemon will reject connections from hosts that have no encryption key defined.
Double quotes within double-quoted strings may be escaped using the backslash character \ (which can be escaped by itself), e.g.:

```
key = "quo\\te me\"."
```

translates to the ASCII sequence

```
quo\te me".
```

Any CIDR range within a host definition needs to to be unique, and the most specific definition will match. The requirement for unambiguousness is quite simply based on the fact that certain host object attributes (key, prompt, enable passwords) may only exist once.

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

```
aaa new-model
tacacs-server host 192.168.0.1 single-connection key your key here
```

The optional `single-connection` parameter specifies that multiple sessions may use the same TCP/IP connection to the server.

Generally, the syntax for host declarations conforms to

`host` = [`realm` *realm*] *name_or_ip-range* { *key-value pairs* }

The optional *realm* tells the daemon to use the host definition for connections associated with a corresponding `spawnd` or `client` realm only. For example,

```
id = spawnd {
  listen = { port = 49 }
  listen = { port = 4949 realm = space }
}

id = tac_plus {
  host = earth { address = ::/0 ... }
  host realm space = mars { address = ::/0 ... }
  group = admin { ... }
  group = alien { ... }
  user = marc { member = admin@earth member = alien@mars }
}
```

makes group membership dependent on the address and/or port the NAS uses to contact the TACACS+ server. User `marc` will be a member of group `admin` for network access servers connecting via port 49, and member of `alien` for those using port 4949.

In most cases, hosts can be refered to either by IP address or by name.

The key-value pairs permitted in host sections of the configuration file are explained below.

- `key` [`warn` [(*YYYY-MM-DD* | *s*)] = *string*

  This sets the key used for encrypting the communication between server and NAS. Multiple keys may be set, making key migration from one key to another pretty easy. If the `warn` keyword is specified, a warning message is logged when a NAS actually uses the key. Optionally, the `warn` keyword accepts a date argument that specifies when the warnings should start to appear in the logs.

  During debugging, it may be convenient to temporarily switch off encryption by using an empty key:

```
key = ""
```

Be careful to remember to switch encryption back on again after you've finished debugging.

- `usage = ( all | any | client | server | group-membership | none )`

  If set to `client` or `none`, TACACS+ session requests that originate from addresses matching the host object will be rejected. Likewise, if set to `server` or `none`, logins that originate from addresses matching the host object will be refused. If `group-membership` is set, the host definition may be used for group membership assignments only.

  This setting is inherited from a supernet to its subnets and defaults to `any`.

  Realms

  You can combine `usage` with *realms* to strictly separate client and server host definitions:

```
# A "clients"-realm that summarizes valid client addresses for authentication:
realm = clients {
  host = ::0/0 { usage = none }
  host = 172.17.2.0/24 { usage = clients }
}

# Use the "clients" realm for looking up NACs:
client realm = clients # This is the global definition, but ...

# ... the "client realm" setting can be used on a per-host basis, too:
host = router { ... client realm = clients ... }
```

- `inherit = ( yes | no )`

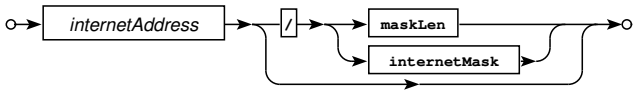
  If set to `no`, the current host declaration won't inherit configuration snippets (e.g.: key, enable passwords) from its supernet. Default is `yes`.

- `name =` *string*

  Assigns the name *string* to the host object. This, combined with the `address` keyword, may be used to form host groups. Don't use this attribute if you've already assigned a name in the initial `host =` statement, and take care to define hosts *before* users and groups, or group membership assignments may fail, possibly silently.

- `address =` *cidr*

  Adds the address range specified by *cidr* to the current host definition.



Railroad diagram: CIDR

- `address file =` *file*

  Add the addresses from *file* to the current host definition. Shell wildcard patterns are expanded by `glob(3)`.

- `client realm =` *realm*

  Lookup NAC parameters in realm *realm* instead of using the `default` one.

- `dns reverse-lookup = ( yes | no )`

  This directive enables or disables DNS reverse-lookups for the corresponding hosts.

- `dns timeout =` *Seconds*

  This directive specifies the maximum time to wait when doing DNS reverse lookups. (Default: `0`).

- `single-connection ( may-close ) = ( yes | no )`

  This directive may be used to permit or deny the single-connection feature for a particular host object. The `may-close` keyword tells the daemon to close the connection if it's unused.

  > **Caveat Emptor**
  > There's a slight chance that single-connection doesn't work as expected. The single-connection implementation in your router or even the one implemented in this daemon (or possibly both) may be buggy. If you're noticing weird AAA behaviour that can't be explained otherwise, then try disabling single-connection on the router.

- `template =` (*address* | *name*)

  This uses part of an existing host definition for *address* or *name* as template for the current host declaration. Only `banners`, `key`, `enable` passwords and `content` are copied, and no recursive lookups take place.

4.3.3.1 Timeouts

The connection timeout may be specified:

- `connection timeout =` *s*

  Terminate a connection to this NAS after an idle period of at least *s* seconds. Defaults to the global option.

4.3.3.2 Authentication

The following authentication related directives are available at host object level:

- `aaa realm =` *realmName*

  Use the aaa configuration (users, groups, mavis, ...) from realm *realm* instead of the `default` one.

- `access acl = ( [ permit ] | deny ) [ not ]` *acl*

  Authentication requests not matching this directive will be rejected.

- `password ( max-attempts =` *integer* | `backoff =` *seconds*)

  The `max-attempts` option sets the maximum number of `Password:` prompts per session. Likewise, `backoff` specifies the (host-specific) delay before returning a `Password incorrect.` message and closing the session.

  Setting these parameters here overrides the corresponding global options, which come with a more exhaustive explanation.

  `backoff` can be specified for both NAC and NAS host objects. If both are given, the smaller one wins.

- `pap password mapping = ( login | pap )`

  When set to `login`, PAP authentication requests will be mapped to ASCII Login requests. You may wish to uses this for NEXUS devices.

- `default user =` name

  This directive sets a default user name for connections sourced from hosts matching this host object. For example:

  ```
  host = nms { address = 1.2.3.4 user = nmsuser }
  ```

  will tell the daemon not to query for the user name for connections coming from `1.2.3.4`, but to use `nmsuser` instead.

  Use this option with care. It may come handy while transitioning to a TACACS+ environment, but has several severe implications, the first one being that the router has no idea about the username, which implies that authorization via TACACS+ won't work.

- `default group =` name

  For users without any group membership this directive may be used to assign one; it's used for connections originating from this host object. For example, you can deny access to user without group membership by using something like that:

```
host = ... { default group = no_login }
acl script = no_login { deny }
group = no_login { acl = no_login }
```

- `enable` [*level*] `=` (`permit` | `deny` | `login` | (`clear` | `crypt`) *password*)

  This directive may be used to set host specific enable passwords, to use the *login* password, or to permit (without password) or refuse any enable attempt. *level* defaults to 15.

  Enable passwords specified at host level have a lower precedence as those defined at user or group level.

  Password Hashes
  You can use the `openssl passwd` utility to compute password hashes. For example,

  ```
  openssl passwd -1 clear_text_password
  ```

  returns a MD5 hash.

  You can enable via TACACS+ by configuring on the NAS:

```
aaa authentication enable default group tacacs+ enable
```

- `anonymous-enable` = (`permit` | `deny`)

  Several broken *TACACS+* implementations send no or an invalid username in `enable` packets. Setting this option to `deny` enforces user authentication before enabling. Setting this option here has precedence over the global option.

- `augmented-enable` = (`permit` | `deny`)

  For *TACACS+* client implementations that send `$enable$` instead of the real username in an enable request, this will permit user specific authentication using a concatenation of username and login password, separated with a single space character. Setting this option here has precedence over the global option.

  `enable` [*level*] `= login` needs to be set in the users' profile for this option to take effect.

4.3.3.3 Authorization

The following authorization related directives are available at host object level:

- `permit if-authenticated` = (`yes` | `no`)

  This will cause authorization for users unknown to the daemon to succeed (e.g. when logging in locally while the daemon is down or while initially configuring TACACS+ support and messing up).

4.3.3.4 Banners and Messages

The daemon allows for various banners to be diplayed to the user:

- `welcome banner` = *string*
- `motd banner` = *string*
- `failed authentication banner` = *string*

  The `failed authentication banner` gets displayed upon final failure of an authentication attempt.

- `reject banner` = *string*

  The `reject banner` gets displayed in place of the welcome message if a connection was rejected by an `access` ACL defined at host, user or group level.

- `message` = *string*

The time when those texts get displayed largely depends on the actual login method:

Context	Directive	Login via Telnet	Login via SSHv1	Login via SSHv2
`host`	`welcome banner`	displayed before `Username:`	not displayed	displayed before `Password:`
`host`	`reject banner`	displayed before closing connection	not displayed	not displayed
`host`	`motd banner`	displayed after successful login	not displayed	displayed after successful login
`host`	`failed authentication banner`	displayed after unsuccessful login	not displayed	displayed after unsuccessful login
`user` or `group`	`message`	displayed after `motd banner`	not displayed	displayed after `motd banner`

Neither the `motd banner` nor a `message` defined in the users' profile will be displayed if `hushlogin` is set for the user.

Both banners and messages support `strftime(3)`-style conversion specifications, plus the following conversion sequences:

`%%r`	router address
`%%R`	router DNS name (fallback: IP address)
`%%c`	client address
`%%C`	client DNS name (fallback: IP address)
`%%p`	router port
`%%u`	username
`%%%`	literal `%`

Example:

```
host = ... {
  ...
  welcome banner = "Welcome. Today is %A.\n"
  ...
}
```

4.3.3.5 Workarounds for Client Bugs

The directive

`client bug` = *value*

may improve compatibility with clients that violate the protocol. Currently, the following bit values (yes, you can use bitwise OR here) are recognized:

Bit	Value	Description
0	1	Ignore invalid AUTHEN/START data, seen with ZTE devices that put their MAC address there.
1	2	Accept version 1 for authorization and accounting packets, seen with Palo Alto systems.

Example:

```
host = ... {
  ...
  client bug = 2
  ...
}
```

4.3.3.6 Inheritance and Hosts

For host lookups, the daemon looks for the most specific host definition first. If that definition (or the requested value) doesn't exist, it looks for the next less-specific one. This process continues through the entire (IPv6) CIDR hierarchy, until a match is found.

In other words: For host parameters, the most specific host definition with that parameter defined matches.

For example, if a host `192.168.1.15` connects to the server, the server needs to know which encryption key to use.

In IPv6-mapped CIDR notation (which the daemon uses internally), the IPv4 CIDR range

```
192.168.1.15/32
```

equals

```
0000:0000:0000:0000:0000:FFFF:C0A8:010F/128
```

The daemon will search for a key in the host definition for `192.168.1.15` and the (defined) supernets:

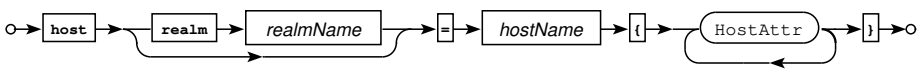
```
0000:0000:0000:0000:0000:FFFF:C0A8:010F/128
0000:0000:0000:0000:0000:FFFF:C0A8:010F/127
0000:0000:0000:0000:0000:FFFF:C0A8:010E/126
0000:0000:0000:0000:0000:FFFF:C0A8:010C/125
...
0000:0000:0000:0000:0000:FFFF:C000:0000/98
0000:0000:0000:0000:0000:FFFF:8000:0000/97
0000:0000:0000:0000:0000:FFFF:0000:0000/96
0000:0000:0000:0000:0000:FFFE:0000:0000/95
0000:0000:0000:0000:0000:FFFC:0000:0000/94
...
0000:0000:0000:0000:0000:0000:0000:0000/3
0000:0000:0000:0000:0000:0000:0000:0000/2
0000:0000:0000:0000:0000:0000:0000:0000/1
0000:0000:0000:0000:0000:0000:0000:0000/0
```

The first key found will be used, as it is the most specific one.

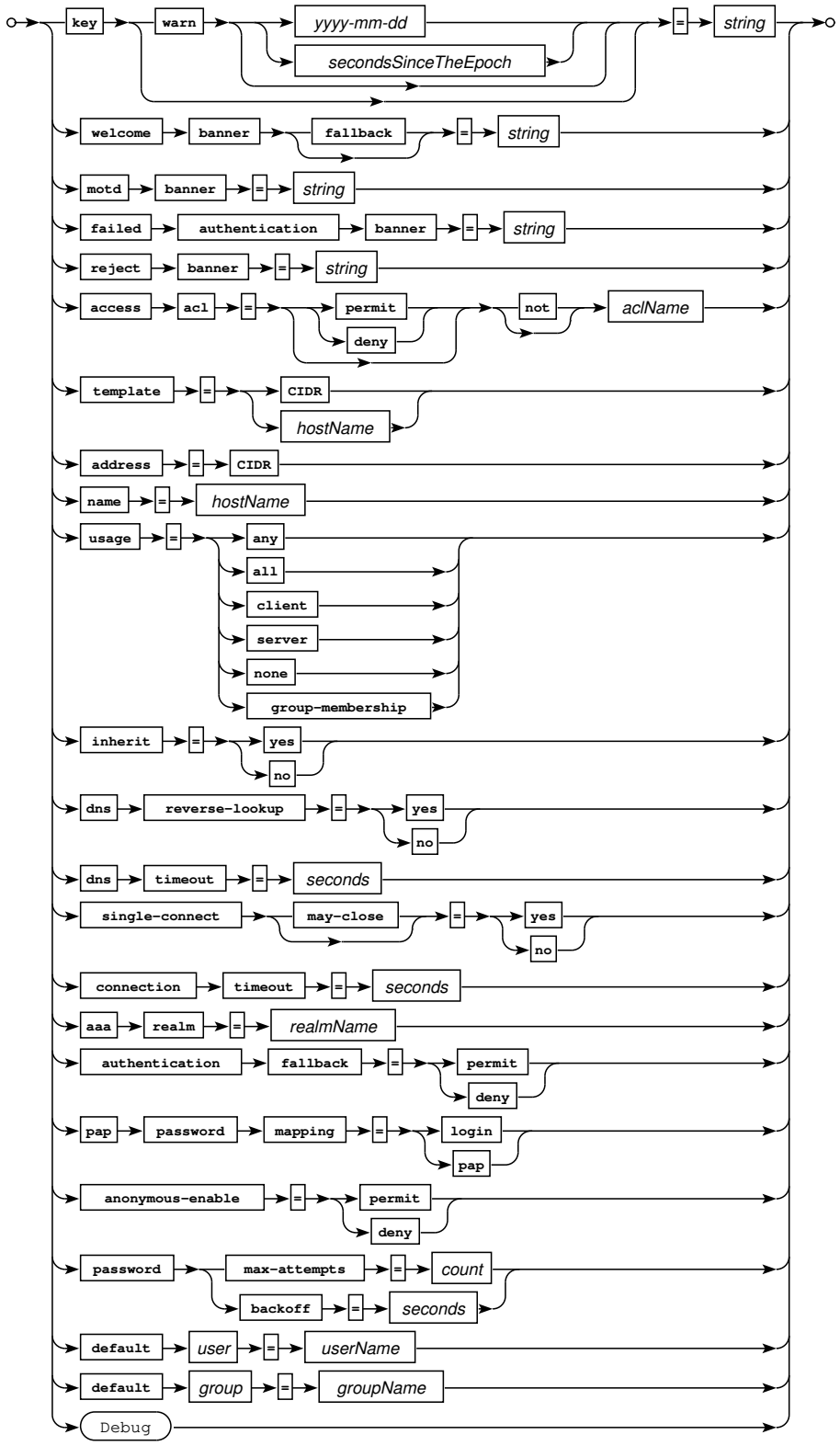
By default, `welcome banner`, `motd banner`, `key`, `enable`, `anonymous-enable` and `content` specifications are inherited. Inheritance may be switched off for a host object:

```
    inherit = no
```

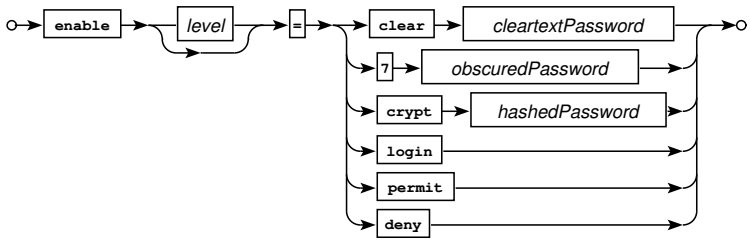
4.3.3.7 Railroad Diagrams



Railroad diagram: HostDecl



Railroad diagram: HostAttr



Railroad diagram: EnableExpr

4.3.3.8 Example

```
host = 10.0.0.0/8 {
    name = customer1
    key = "your key here"
    welcome banner = "\nHitherto shalt thou come, but no further. (Job 38.11)\n\n"
    enable 15 = clear whatever
}

host = test123 {
    address = 10.1.2.0/28
    address = 10.12.1.30/28
    address = 10.1.1.2
    # key/banners/enable will be inherited from 10.0.0.0/8 by default,
    # unless you specify "inherit = no"
    address file = /some/path/test123.cidr
    prompt = "\nGo away.\n\n"
}
```

4.3.4 Time Ranges

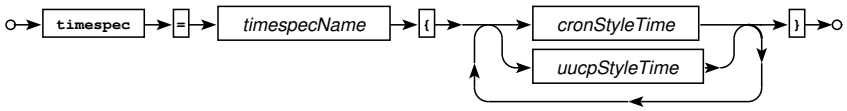
`timespec` objects may be used for time based profile assignments. Both `cron` and `Taylor-UUCP` syntax are supported; see you local `crontab`*(5)* and/or UUCP man pages for details. Syntax:

`timespec` = *timespec_name* { "*entry*" [...] }

Example:

```
# Working hours are from Mo-Fr from 9 to 16:59, and
# on Saturdays from 9 to 12:59:
timespec = workinghours {
    "* 9-16 * * 1-5"   # or: "* 9-16 * * Mon-Fri"
    "* 9-12 * * 6"     # or: "* 9-12 * * Sat"
}

timespec = sunday { "* * * * 0" }

timespec = example {
    Wk2305-0855,Sa,Su2305-1655
    Wk0905-2255,Su1705-2255
    Any
}
```

4.3.4.1 Railroad Diagrams



Railroad diagram: TimespecDecl

4.3.5 Access Control Lists

Access Control Lists (ACLs) consist of a series of Access Control Expressions (ACEs). An ACL does match whenever one of its ACEs does. The ACEs that form an ACL are processed in sequence. An ACE matches if at least one of each defined subtype expression type matches. ACEs named identially become part of an ACL with the same name.

Standard ACL syntax is:

`acl` = *name* { ((`nac` (= [`not`] (*HostName*|*cidr*)) | ([`dns`] `regex` = [`not`] *NacRegex*)) | (`nas` (= [`not`] (*HostName*|*cidr*)) | (`regex` = [`not`] *NasRegex*)) | (`port` `regex` = [`not`] *PortRegex*) | (`acl` = [`not`] *ACLName*)* } (`time` = [`not`] *TimeSpecName*)* }

Alternatively, scripting can be used:

`acl script =` *name* { ... }

The latter is detailled in the next chapter.

For standard ACL syntax, please have a look at the Railroad Diagrams below, and things will hopefully become much clearer.

ACEs will be evaluated in the order you've defined them. E.g.,

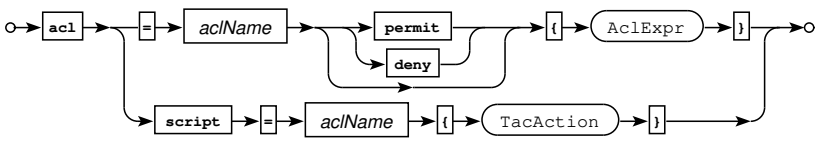
```
acl = myacl {
    nac = 10.0.0.0/8
    nac = 11.0.0.0/8
    nas = somehostobjectname
}

acl = myacl {
    time = workinghours
}
```

consists of two ACEs which form an ACL named "myacl". This ACL matches

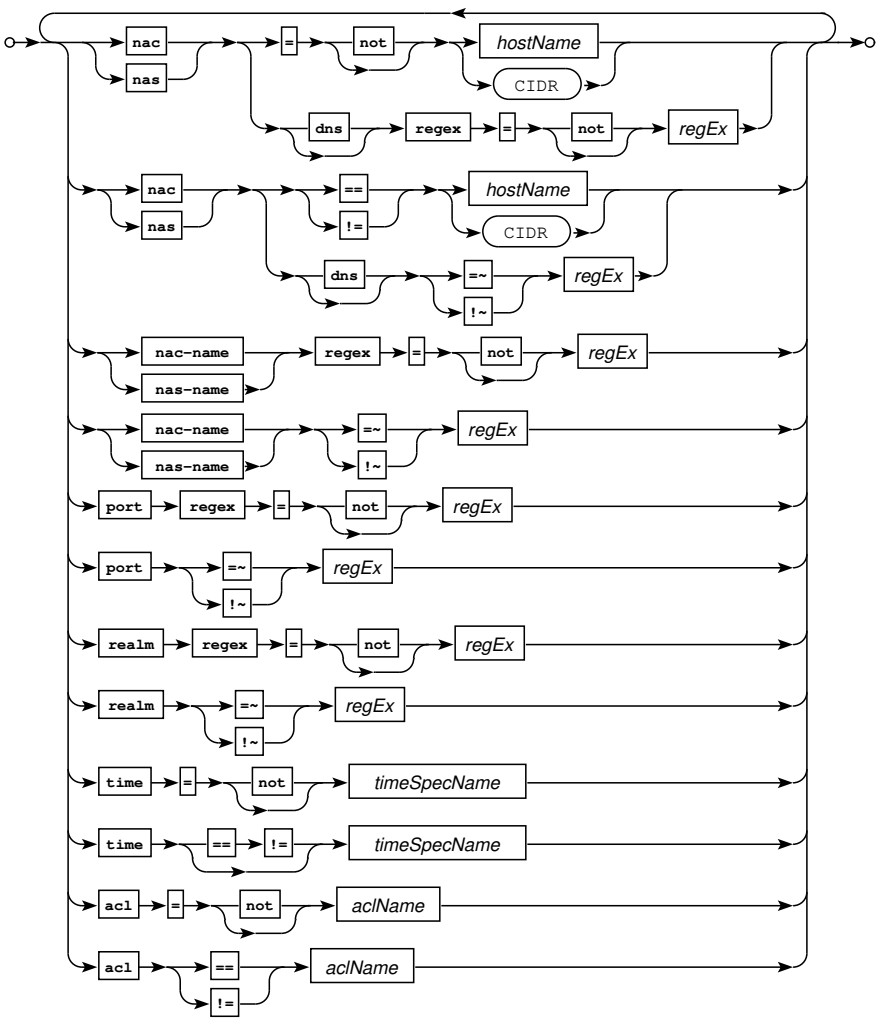
- for clients in 10.0.0.0/8 or in 11.0.0.0/8, and servers part of `somehostobjectname`
- or simply during `workinghours`, no matter what clients or servers (or, for that matter, ports) are involved.

When using the `dns` keyword, NAC regex matching is done against the NACs DNS reverse mapping. This will only work if the software was compiled with `c-ares` support. However, keep in mind that DNS isn't necessarily trustworthy and lookups might plain fail. Keep in mind that you're best off enabling `single-connection` on your NAS. This allows the daemon to take advantage of caching the reverse-mapping results.

4.3.5.1 Railroad Diagrams



Railroad diagram: ACLDecl



Railroad diagram: ACLExpr

4.3.6 Rewriting User Names

This is experimental. It requires the binary to be built with PCRE v2 support (using the `--with-pcre2` configure option).

A host may refer to a rewrite profile defined at realm level to substitute user names. The following sample code will map both `admin` and `root` to `marc`, and convert all other usernames to lower-case:

```
rewrite = rewriteRule {
    rewrite /^admin$/ marc
    rewrite /^root$/ marc
    rewrite /^.*$/ \L$0
}

host = ... {
    ...
    rewrite user = rewriteRule
    ...
}
```

Please keep in mind that this is experimental ...

4.3.7 Scripts

Scripts may currently be used for ACLs and in service declaration scope:

- `acl script =` acl_name `{` tac_action ... `}`

  Example:

```
acl script = myacl123 {
    if (nas == 1.2.3.4 || nac = SomeHostName || nac-dns =~ /\\.example\\.com$/) deny
}
```

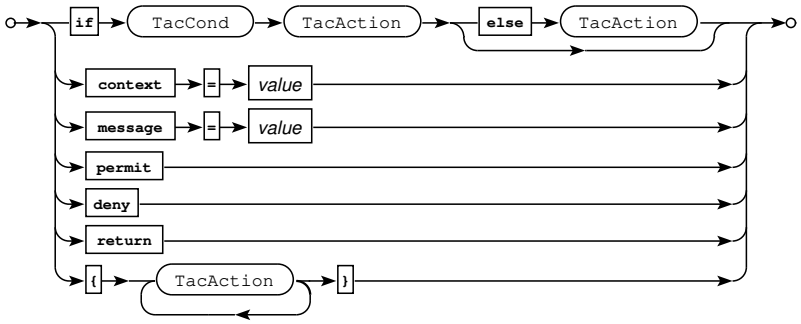
- `script = {` tac_action ... `}`

  Example:

```
user = joe {
    service = shell {
        script = {
            if (cmd == "") permit # required for shell startup
            if (cmd =~ /^(no\s)?shutdown\s/) permit }
    }
}
```

4.3.7.1 Syntax

A script consists of a series of actions:

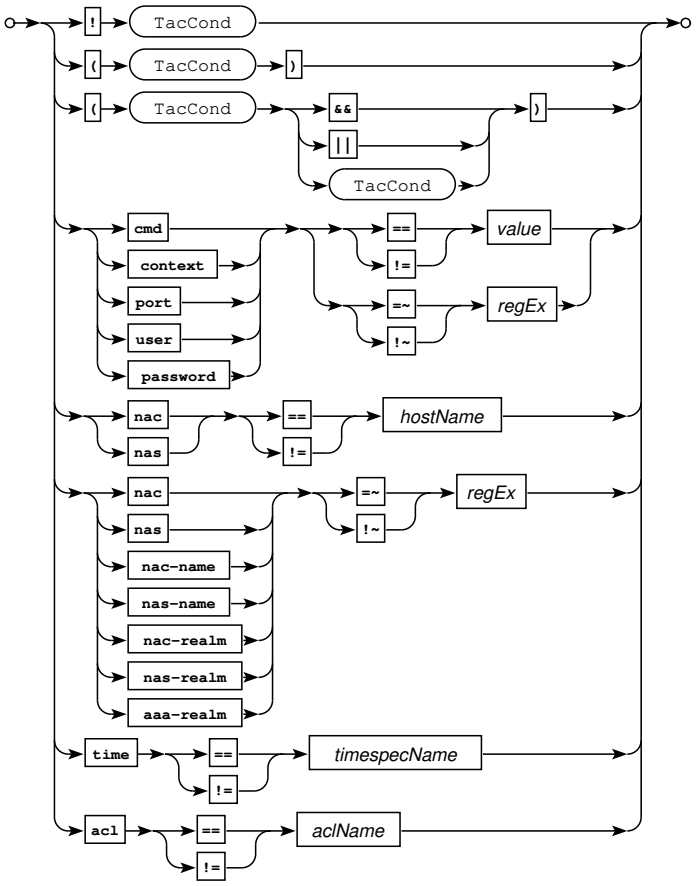


Railroad diagram: TacAction

The actions `return`, `permit` and `deny` are final. At the end of a script, `return` is implied, at which the daemon continues processing the configured `cmd` statements in `shell` context) or standard ACLs (in ACL context). The assignment operations (`context =`, `message =`) do make sense in `shell` context only.

Setting the `context` variable makes sense in `shell` context only. See the example in the corresponding section.

Condition syntax is:



Railroad diagram: TacCond

`cmd` and `context` may be used in `shell` context only.

4.3.8 Users and Groups

User and group declarations are pretty similar. The major difference is, that groups are *static*, while user declarations may be added at run-time via the *MAVIS* back-end.

A user may be member of a group (also known as *role* or *profile*). Actual group membership may depend on various factors, e.g. the NAS the user is on, the NAC, or time ranges. Each group may in turn be member of another group and so on, ad infinitum.

The basic form of a user and group declarations is

```
user = username { ...  }
```

or, respectively,

```
group = groupname {  ...  }
```

A user or group declaration may contain key-value pairs and service declarations.

4.3.8.1 User-only options

The following declarations are valid in *user* context only:

- `login = ( ( clear | crypt ) password | mavis | permit | deny )`
  The `login` password authenticates `shell` log-ins to the server.

```
login = crypt aFtFBT4e5muQE
login = clear Ci5c0
```

For `crypt`, DES- or MD5-hashed passwords may be used.

If the `mavis` keyword is used instead, the password will be looked up via the **MAVIS** backend. It will not be cached. This functionality may be useful if you want to authenticate at external systems, despite static user declarations in the configuration file.

- `pap =` ((`clear` | `crypt`) *password* | `mavis` | `permit` | `deny`)

  The `pap` authenticates PAP log-ins to the server. Just like with `login`, the password doesn't need to be in clear text, but may be DES (or MD5) hashed, or may be looked up via the **MAVIS** backend.

- `arap =` (`clear`*password* | `permit` | `deny`)

  For ARAP authentication, a cleartext password is required.

- `chap =` (`clear`*password* | `permit` | `deny`)

  For CHAP authentication, a cleartext password is required.

- `ms-chap =` (`clear`*password* | `permit` | `deny`)

  For MS-CHAP authentication, a cleartext password is required.

- `password =` (`clear`*password* | `permit` | `deny`)

  This directive sets all *previously undefined* passwords to the given cleartext password. This directive is retained for backwards compatibility only, and usage is deprecated.

- `password` [`acl` [`not`] *acl*] `{ ... }`

  This directive allows specification of ACL-dependent passwords. Example:

```
acl jumpstation = { nac == 10.255.0.85 }

user = marc {
    password acl jumpstation {
        login = permit
        pap = permit
    }
    password {
        login = clear myLoginPassword
        pap = clear myPapPassword
    }
}
```

4.3.8.2 Group-only options

The following declarations are valid in *group* context only:

- `template =` *groupname*

  This directive merges group definitions from *groupname* to the current group. Already existing definitions take precedence.

4.3.8.3 User and Group options

The following key-value pairs are valid in both *user* and *group* context:

4.3.8.3.1 Limits

Validity of an user or group may be limited by various attributes:

- `acl =([permit]|deny)[not]` *acl*

  Validity of of a user or group profile may be restricted by ACLs. Multiple ACLs may be specified and are evaluated in order.

- `valid from =` (*YYYY-MM-DD*|*s*)

  The profile will be valid starting at the given date, which can be specified either in ISO8601 date format or as in seconds since January 1, 1970, UTC.

- `valid until =` (*YYYY-MM-DD*|*s*)

  The profile will be invalid after the given date.

- `client[regex] = (deny|[permit])` *string* `[, ...]`

  As an alternative (or complement) to ACLs (see above), certain NAC based validity restrictions may be placed in a user (or group) object directly. For example, to limit log-ins to a certain IP range or a host group:

```
client = 10.99.0.0/16
client = deny customer3
```

  If the `regex` keyword is used, *string* is expected to be a valid regular expression. Regular expression search is expensive and only used if the NAC address sent by the NAS ist *not* an IP address. Example for regular expression client matching:

```
# if compiled with PCRE:
client = deny /^async$/
client regex = /^console$/

# without PCRE:
client = deny "^async$"
client regex = "^console$"
```

> **Evaluation Order**
> Please keep in mind that ACLs are checked *before* `client` statements.
> For `client` definitions, IP addresses are checked *before* regular expressions, while the latter are evaluated in the order you've defined them.

- `server = (deny|[permit])` *string* `[, ...]`

  This limits user account validity to certain NAS addresses or groups, e.g.:

```
server = 1.2.3.4/32
server = nas01
```

4.3.8.3.2 Messages

These options handle displaying of messages to the user.

- `message =` *string*

  A message displayed to the user upon log-in.

- `hushlogin =(yes|no)`

  Setting `hushlogin` to `yes` keeps the daemon from displaying `motd` and `user` messages upon login.

4.3.8.3.3 Enable passwords

Enable passwords may be specified at, in order of preference, host, user or group level:

- `enable` [*level*] `=` (`permit` | `deny` | `login` | (`clear` | `crypt`) *password*)

  This directive may be used to set user or group specific enable passwords, to use the *login* password, or to permit (without password) or refuse any enable attempt. Enable secrets defined at group or user level have precedence over those defined at host level. *level* defaults to 15.

  The default privilege level for an ordinary user on the NAS is usually 1. When a user enables, she can reset this level to a value between 0 and 15 by using the NAS `enable` command. If she doesn't specify a level, the default level she enables to is 15.

4.3.8.3.4 MAVIS options

The following *MAVIS* related option is available for groups and locally defined users:

- `mavis realm` = *realmName*

  For locally defined users with `password = mavis` (or one if its variations) this directive selects the realm to use for *MAVIS* authentications.

4.3.8.3.5 Group membership

Group membership is specified using the `member` directive:

- `member` [`acl` [`not`] *acl*] = *string* [`,` ...]

  This directive defines group membership, optionally depending on the NAS server's address or on some access list. For example, you could configure:

```
host = campus { ... }
host = remote { ... }
host = router { ... }
timespec = workinghours { ... }
group = admins { ... }
group = staff { ... }
group = guest { ... }
```

  plus a couple of ACLs:

```
acl oncampus = { nac = campus }
acl viavpn = { nac = remote }
acl atwork = { nac = campus time = workinghours }
```

  and then base group membership on the latter:

```
user = adminuser { member = admins }
user = employee {
    ...
    member acl atwork = staff@router
    member acl viavpn = staff
    member acl remote = guest
    ...
}
user = guest {
    ...
    member acl atwork = guest
    ...
}
user = myself {
    ...
```

```
    member = admins@router
    member = staff
    ...
}
```

`member` directives are to be ordered by ACL name, with definitions without ACL coming last. This is enforced at parsing time.

Alternate groups may be specified, with group names separated by `/`:

```
member = dialin/callback@10.11.12.13
```

Here, group membership will default to the `dialin` group. However, appending the *separation character* `*` (changeable using the `separation tag =` *character* directive), followed by `callback` to the username *at login time* will choose the corresponding `callback` group profile instead.

To iterate this again: Any user can be a member of exactly one primary group. The `dialin/callback` syntax in this example actually means: The user is free to choose at *login time* whether he wants to be a member of `dialin` XOR `callback`. User `fred` may authenticate as `fred` (or as `fred*dialin`) to gain `dialin` membership, and `fred*callback` will assign him to the `callback` group.

The users' choice can be overridden using the `tag` directive (see below).

Group membership may depend on NAS IP, e.g. `fred` could have something like

```
member = test1@nas1
member = test2@nas2
```

in his profile.

Just like users, groups can be member of (other) groups:

```
group = test1 { ... }
group = test2 { ... }

group = test2 {
   ...
   member = test1
   member = test2@nas1
   ...
}
```

Simultaneous membership in (non-hierarchical) groups isn't supported.

As group membership is resolved when parsing the configuration file, groups and hosts need to be already defined before being used as argument in a `member` definition.

- `nas default restriction =` (*cidr* | *host*)

  Limits usage of the group to the specified hosts, unless specified otherwise. Example:

```
group = group1 { nas default restriction = host1 }
group = group2 { }
group = group3 { nas default restriction = host1 }
user = user1 { member = group1,group2,group3@host3 }
```

  is identical to

```
group = group1 { }
group = group2 { }
group = group3 { }
user = user1 { member = group1@host1,group2,group3@host3 }
```

  Note: `nas default restriction` will be ignored for alternate group specifications (e.g. `member = group1/group2`).

- `tag` (`acl` [`not`] *acl*) = *string*

  As described above, a user may manually select an (allowed) group profile at login time using the `user*group` syntax. You can override the user's choice using the `tag` directive in user or group profiles. Plus, you can make acl-based tag selections. E.g.:

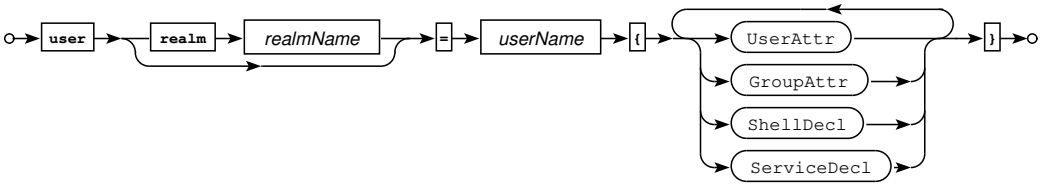
```
timespec = workinghours { "* 9-16 * * 1-5" "* 9-12 * * 6" }
timespec = weekend { "* * * * 6-1" }
timespec = changewindow { "* 22-23 * * Sat" "* 0-3 * * Sun" }

acl helpdesk = {
    time = workinghours
    nac = internal_lan
    nas = somecustomer
}

acl remoteadmin = {
    time = changewindow
    nac = dialin
}

user = fred {
    login = clear test
    member = admin/ro/emergency
    tag acl helpdesk = admin
    tag acl remoteadmin = emergency
    tag = ro
}
```

  `fred` will be a member of the `admin` group during work hours, but only if his client IP address is part of the internal LAN and if he's on a certain NAS. He'll be member of the `emergency` group during change window hours, but needs to come from the dial-in pool. Outside these hours he's a member of the `ro` hours.

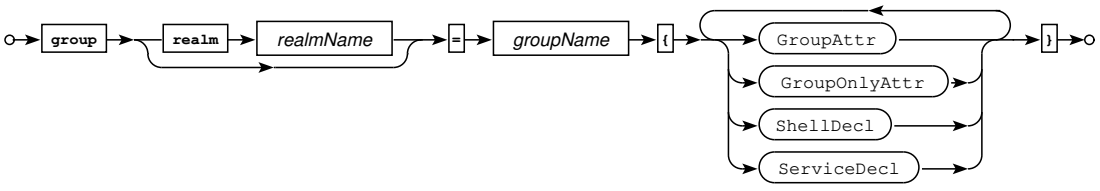
  `tag` definitions are evaluated in order. If no definition matches, the user's session tag (the group selected at login time using `user*group` syntax) is used. If no group tag was given, the first group in the `member` definition is selected.

  In most cases, it might be a better idea to use `member acl` syntax instead of `tag acl`.

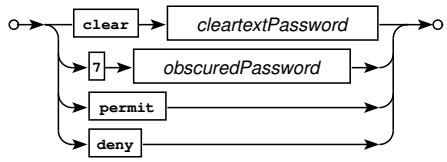
4.3.8.3.6 Railroad Diagrams



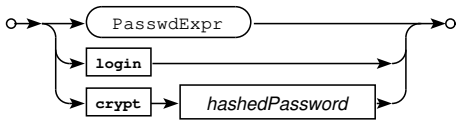
Railroad diagram: UserDecl



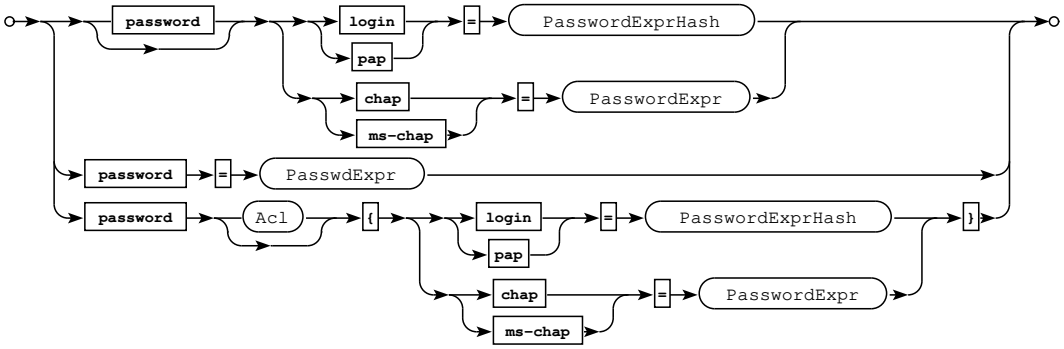
Railroad diagram: GroupDecl



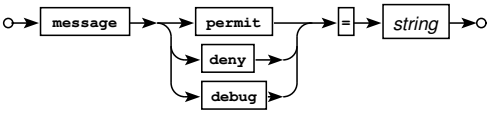
Railroad diagram: PasswordExpr



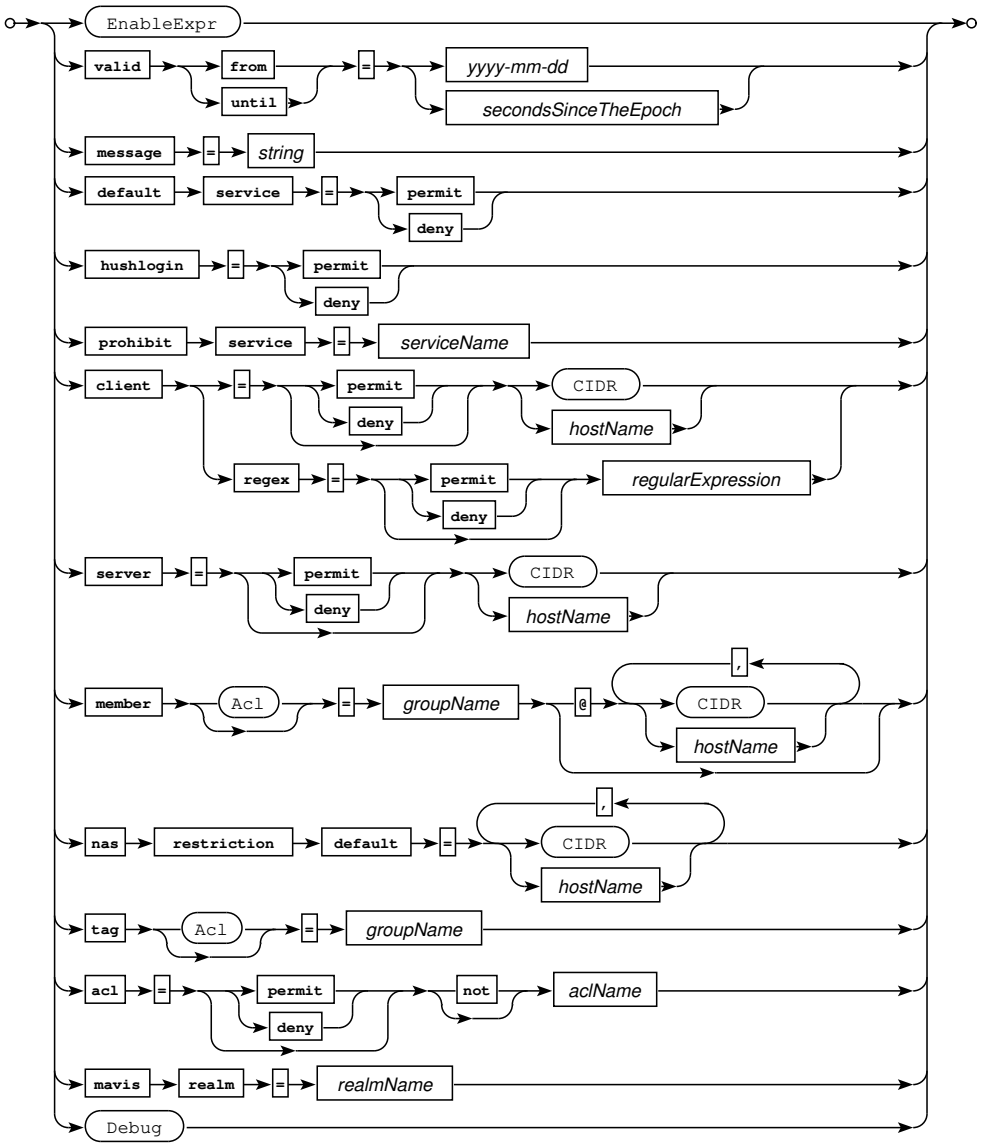
Railroad diagram: PasswordExprHash



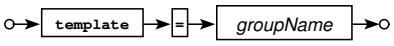
Railroad diagram: UserAttr



Railroad diagram: UserMessage



Railroad diagram: GroupAttr



Railroad diagram: GroupOnlyAttr

4.3.8.4 Service Restrictions

- `default service = (permit|deny)`
  This defines whether a service not explicitely defined in the user profile should be permitted or denied.
- `prohibit service =` *service ...*
  Use this to override the default service specification. E.g., to explicitely deny the X.25 service:

```
    prohibit service = x25
```

4.3.8.5 Service Definitions

Service definitions may appear in *user* and *group* sections. Services can easily be made dependent on ACLs:

`service` (`acl` [`not`] ACLName = *service* { ... }

There are a couple of generic configuration attributes which may appear in arbitrary service definitions. In particular:

- `default attribute =` (`permit` | `deny`)
  This directive specifies whether the daemon is to accept or reject unknown attributes sent by the NAS (default: `deny`).
- `acl =` ([`permit`] | `deny`) [`not`] *acl*
  Just as user and group profiles, services can be restricted by using ACLs.
- (`set` | `add` | `optional`) *attribute* = *value*
  Defines mandatory and optional attribute-value pairs:
  - `set` unconditionally returns a mandatory AV pair to the NAS
  - `optional` returns a NAS-requested (and perhaps modified) optional AV pair to the NAS unless the attribute was already in the mandatory list
  - `add` returns an optional AV pair to the client even if the client didn't request it (and it was neither in the mandatory nor optional list)

  Example:

```
set priv-lvl = 15
```

  For a detailed description on mandatory and optional AV-pairs, see the "The Authorization Algorithm" section somewhere below.
- `return`
  Use the current service definition as-is. This stops the daemon from checking for the same service in the groups the current user (or group) is a member of.

Other configuration attributes are service specific and only valid in certain contexts:

SHELL (EXEC) Service

Shell startup should have an appropriate service

```
service = shell { }
```

defined. Valid configuration directive within the curly brackets are:

- `default cmd =` (`permit` | `deny`)
  This directive specifies whether the daemon is to accept or reject commands not explicitely permitted (default: `deny`).
- `message` (`permit` | `deny`) = *string*
  This specifies a message to be presented to the user on accepting or rejecting a command. Recognized string substitutions within *string* are `%c` for the command name, `%a` for the command arguments and `%C` for the currenly set context. Example:

```
message permit "Permitted '%c %a'"
message deny "Denied '%c %a'"
```

  This directive may appear in `cmd` sections, too, where it overrides the `service` section definitions.
- `cmd =` *command* { ... }
  cmd sections permit or deny commands and command arguments. Example:

```
cmd = show {
    deny /^(running|startup)-config/
    deny tech-support
    permit //
    message deny = "Try this again and your account will be revoked."
    message permit = "This data is to be considered confidentional."
}
cmd = reload {
    permit //
    message permit = "You hopefully know what you're doing ..."
}
```

Regular expression syntax is POSIX or, if enabled at compile time, PCRE.

If you're unsure what commands and arguments the router actually sends for verification, you may simply modify a user (or group) profile to display those within a login session. E.g.,

```
user = ... {
    ...
    debug = CMD
    ...
    service = shell {
        ...
        message debug  = "author: 'cmd = %c { permit /^%a$/ }"
        ...
    }
}
```

will display the configuration snippets to permit the command in PCRE syntax:

```
Router#conf t
author: 'cmd = configure { permit /^terminal <cr>$/ }'

Enter configuration commands, one per line.  End with CNTL/Z.
Router(config)#
```

Non-Shell Services

E.g. for PPP, *protocol* definitions may be used:

```
service = ppp {
    protocol = ip { set addr = 1.2.3.4 }
}
```

Use

```
    default protocol = permit
```

or

```
    default protocol = deny
```

to specify the default for protocols not explicitly defined within a service declaration. (default: `deny`).

For a Juniper Networks-specific authorization service, use:

```
service = junos-exec {
   set local-user-name = NOC
   # see the Junos documentation for more attributes
}
```

Likewise, for Raritan Dominion SX IP Console Servers:

```
service = dominionsx {
    set port-list = "1 3 4 15"
    set user-type = administator # or operator, or observer
}
```

Quotes
If your router expects double-quoted values (e.g. Cisco Nexus devices do), you can advise the parser to automatically add these:

```
service = shell {
    set shell:roles="\"network-admin\""
}
```

and

```
service = shell {
    double-quote-values = yes
    set shell:roles="network-admin"
}
```

are equivalent, but the latter is more readable.

4.3.8.6 Host-group specific services

Services can be configured NAS specific by appending `@`*host* to the service name. Here, *host* is the name of a host object, **not** an IP address or CIDR range. For example, a user's PPP address could be static on one particular NAS, but dynamic everywhere else:

```
host = dialin { address = ... }
user = ... {
    service = ppp { protocol = ip { } }
    service = ppp@dialin { protocol = ip { set addr = 1.2.3.4 } }
}
```

While the same can be achieved using host specific group membership, this adds some flexibility for dynamic authentication backends.

4.3.8.7 CLI Contexts

When used with `single-connection` and being told so, the daemon tries to remember command context. which permits the `shutdown` family of commands for exactly one interface, but denies it for all the others:

```
user = john {
    password = clear doe
    service = shell {
        default cmd = permit
        set priv-lvl = 15
        script = {
            if (cmd == "") permit # shell startup

            if (cmd =~ /^interface FastEthernet 0\/1 /) {
                message = "Context has been set. \"[no] shut\" should work for you."
                context = FE
                permit
            } else if (cmd =~ /^interface/){
                message = "Context has been reset."
                context = ""
```

```
                permit
            }
            if (context == FE) {
                if (cmd =~ /^shutdown/) permit
                if (cmd =~ /^no shutdown/) permit
                deny
            }
        }
        cmd = shutdown { deny . }
        cmd = no { deny /^shutdown/ }
    }
}
```

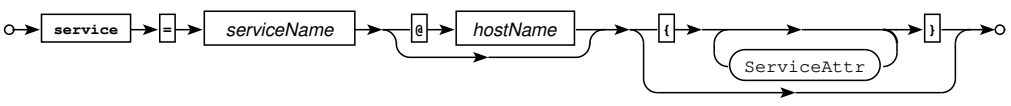
4.3.8.8 Examples

Here we declare two users `fred` and `lily`, and two groups, `admin` and `staff`.

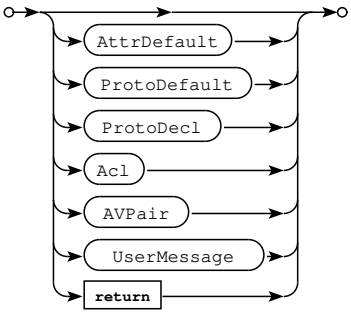
`fred` is a member of group `admin` and group `admin` is in turn a member of group `staff`. `lily` is not a member of any group.

```
group = admin {
    # group admin is a member of group staff
    member = staff
    service = shell {
        set priv-lvl = 15
    }
}

group = staff {
    # group staff 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 on 0.0.0.0/0
    member = admin
    # fred is a member of group staff when logging in on 10.0.0.0/8
    member = staff@10.0.0.0/8
    # fred is a member of group admin when logging in on hosts
    # defined in hostgroup test123
    member = admin@test123
    # fred may only log in from a client in 172.16.0.0/24 ...
    client = 172.16.0.0/24
    # ... or from whatever address is defined in some host object test123
    client = test123
}
```

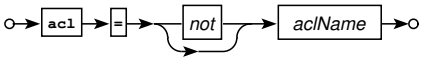
4.3.8.9 Railroad Diagrams



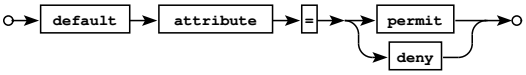
Railroad diagram: ServiceDecl



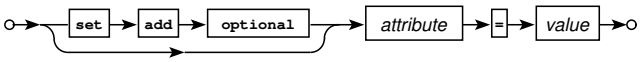
Railroad diagram: ServiceAttr



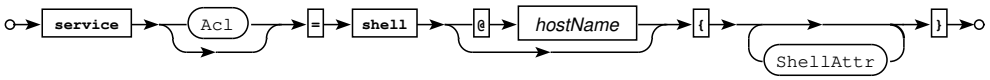
Railroad diagram: Acl



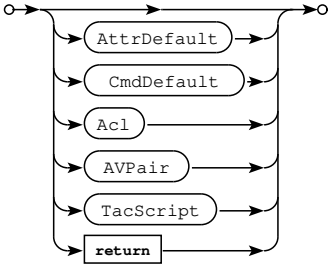
Railroad diagram: AttrDefault



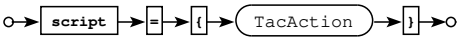
Railroad diagram: AVPair



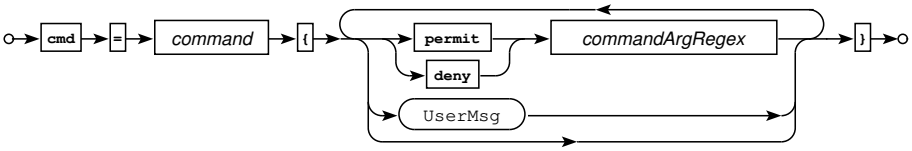
Railroad diagram: ShellDecl



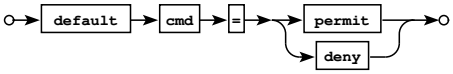
Railroad diagram: ShellAttr



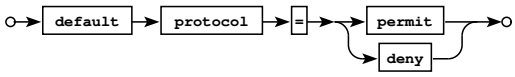
Railroad diagram: TacScript



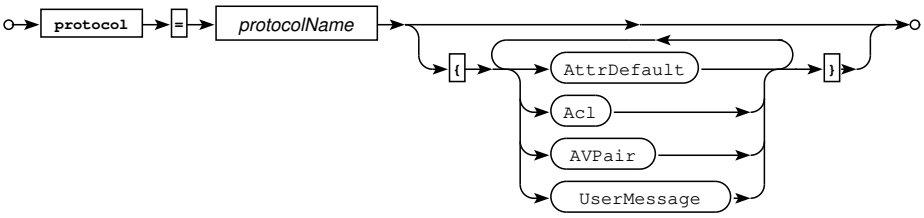
Railroad diagram: ShellCommandDecl



Railroad diagram: CmdDefault



Railroad diagram: ProtoDefault



Railroad diagram: ProtoDecl

4.3.8.10 Configuring Non-local Users via MAVIS

MAVIS configuration is optional. You don't need it if you're content with user configuration in the main configuration file.

MAVIS backends may dynamically create user entries, based, e.g., on LDAP information.

For PAP and LOGIN,

```
pap backend = mavis
login backend = mavis
```

in the global section delegate authentiation to the MAVIS sub-system. Statically defined users are still valid, and have a higher precedence.

By default, MAVIS user data will be cached for 120 seconds. You may change that period using

```
cache timeout = seconds
```

in the global configuration section.

4.3.8.11 Configuring Local Users for MAVIS authentication

Under certain circumstances you may wish to keep the user definitions in the plain text configuration file, but authenticate against some external system nevertheless, e.g. LDAP or RADIUS. To do so, just specify one of

```
    login = mavis
    pap = mavis
    password = mavis
```

in the corresponding user definition.

4.3.8.12 Recursion and Groups

In general, when the daemon looks up values, it will look first to see if the user has her own definition for that value. If not, it looks to see if she belongs to a group and if so, whether the group has a value 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 gmaryroups.

This recursive process occurs for lookups of expiration dates and also for authorization parameters (see later), but not for user passwords.

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.

4.3.8.13 Configuring User Authentication

User Authentication can be specified separately for PAP, ARAP, CHAP, and normal logins. ARAP, CHAP, and global user authentication must be given in clear text.

The following assigns the user mary five different passwords for ARAP, inbound and outbound CHAP, inbound PAP, outbound PAP, and normal login respectively:

```
user = mary {
    arap = clear "arap password"
    chap = clear "chap password"
    pap  = clear "inbound pap password"
    login = crypt XQj4892fjk
}
```

If

```
user backend = mavis
```

is configured in the global section, users not found in the configuration file will be looked up by the MAVIS backend. You should consider using this option in conjuction with the more sophisticated backends (LDAP and ActiveDirectory, in particular), or whenever you're not willing to duplicate your pre-existing database user data to the configuration file. For users looked up by the MAVIS backend,

```
pap backend = mavis
```

and/or

```
login backend = mavis
```

(again, in the global section of the configuration file) will cause PAP and/or Login authentication to be performed by the MAVIS backend (e.g. by performing an LDAP bind), ignoring any corresponding password definitions in the users' profile.

If you just want the users defined in your configuration file to authenticate using the MAVIS backend, simply set the corresponding PAP or Login password field to `mavis` (there's no need to add the `user backend = mavis` directive in this case):

```
user = mary { login = mavis }
```

4.3.8.14 Configuring Expiry Dates

An entry of the form:

```
user = lol {
    valid until = YYYY-MM-DD
    login = clear "bite me"
}
```

will cause the profile to become invalid, starting after the `valid until` date. Valid date formats are both ISO8601 and the absolute number of seconds since 1970-01-01.

A expiry warning message is sent to the user when she logs in, by default starting at 14 days before the expiration date, but configurable via the `warning period` directive.

Complementary to profile expiry,

```
    valid from = YYYY-MM-DD
```

activates a profile at the given date.

4.3.8.15 Configuring Authentication on the NAS

On the NAS, to configure login authentication, try

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

(Alternatively, you can try a *named authentication list* instead of `default`. Please see the IOS documentation for details.)

Don't lock yourself out.
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, you should configure an enable secret and make it the last resort authentication method, so if your TACACS+ daemon fails to respond you will be able to use the NAS enable password to login. To do this, configure:

```
aaa authentication login default group tacacs+ enable
```

or, to if you have local accounts:

```
aaa authentication login default group tacacs+ local
```

If all else fails, and you find yourself locked out of the NAS due to a configuration problem, the section on *recovering from lost passwords* on Cisco's CCO web page will help you dig your way out.

4.3.8.16 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.

4.3.8.17 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 PCRE (Perl Compatible Regular Expressions) or `egrep`-style (POSIX 1003.2) regular expressions to match command arguments (see your local `pcrematching`*(3)* or `regex`*(7)* man page, respectively, for a description of regular expressions), and an action which is `deny` or `permit`.

Case-insensitive pattern-matching
By default, regular expression matching is *case-insensitive*. This can be changed by specifying

```
regex-match-case = yes
```

If a POSIX pattern contains spaces or special characters, you'll have to enclose the pattern in double quotes (`"`). For text inside double quotes, C parsing rules apply. In particular, the backslash `\` needs to be escaped as `\\`. PCRE patterns need to be enclosed in slashes (`/`*pattern*`/`). Both POSIX and PCRE syntax may be used in the same configuration file.

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

```
telnet 131.108.13.any_number
telnet 128.any_number.12.3
show running-config interface anything
```

All other commands are denied (by default).

```
user=fred {
    # default service = deny
    service = shell {
        # default cmd = deny
        cmd = telnet {
            # permit specified telnets
            # POSIX:
            permit ^131\.108\.13\.[0-9]+$
            # PCRE:
            permit /^128\.\d+\.12\.3$/
        }
        cmd = show {
            # permit show commands
            # PCRE:
            permit /^running-config interface/
        }
    }
}
```

Whitespace
If any non-PCRE argument list you specify contains spaces or tabs, you must enclose it in double quotes. PCRE expressions are to be enclosed in slashes, anyway.

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

Matches are only anchored if you specify so, so `deny 131.108.13.[0-9]+` matches anywhere in the command. To anchor the match, use ^ at the beginning of the regular expression.

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.

4.3.8.17.1 Command Expansion

For command authorization, the Cisco NAS expands all commands to their full names e.g. when you type `config t` on the NAS, this will be expanded to `configuration terminal` before being sent to the daemon so that you don't need to list all the possible contractions of a command.

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:

```
user = lol {
    default service = permit
}
```

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

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.

```
user = lol { service = shell { 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 behavior.

4.3.8.18 Authorizing EXEC (SHELL) Startup

You can supply some parameters whenever an shell starts, e.g. an autocommand, 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 = shell {
        # When an exec is started, its connection access list will be 4.
        # It also has an autocmd.
        set acl = 4
        set autocmd = "telnet foobar"

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

Note: specifying an autocommand, or any other exec services, is part of EXEC AUTHORIZATION. For it to work, you must also configure exec authorization on your NAS e.g.

```
aaa authorization exec authorlist group tacacs+ local
aaa authorization commands 15 cmdlist group tacacs+ local
```

and, eventually, on the lines:

```
authorization commands 15 cmdlist
authorization exec authorlist
```

Just as with authentication, you can use a *named authorization list*, or simply `default`.

4.3.8.19 Authorizing EXEC, SLIP, PPP and ARAP services

Authorizing EXEC, SLIP, PPP and ARAP services is done quite differently from command authorization.

When authorizing these 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 $\star$ instead of a = sign, meaning that the value in a pair is optional. An = sign indicates a mandatory value. A $\star$ 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:

```
service=ppp
protocol=ip
addr*131.108.12.44
```

You can use the NAS debugging command

```
debug aaa authorization
```

to see what authorization AV pairs are being used by the NAS. Note: If you are not on the router console, you will also need to issue a `terminal monitor` command to see debug output.

4.3.8.20 The Authorization Process

Authorizing a single session can result in multiple requests being sent to the daemon. For example, in order to authorize a dialin PPP user for IP, the following authorization requests will be made from the NAS:

1. An initial authorization request to startup PPP from the exec, using the AV pairs `service=ppp`, `protocol=ip`, will be made (Note: this initial request will be omitted if you are autoselecting PPP, since you won't know the username yet).

   This request is really done to find the address for dumb PPP (or SLIP) clients who can't do address negotiation. Instead, they expect you to tell them what address to use before PPP starts up, via a text message e.g. "Entering PPP. Your address is 1.2.3.4". They rely on parsing this address from the message to know their address.

2. Next, an authorization request is made from the PPP subsystem to see if PPP's LCP layer is authorized. LCP parameters can be set at this time (e.g. callback). This request contains the AV pairs `service=ppp`, `protocol=lcp`.

3. Next an authorization request to startup PPP's IPCP layer is made using the AV pairs `service=ppp`, `protocol=ipcp`. Any parameters returned by the daemon are cached.

4. Next, during PPP's address negotiation phase, each time the remote peer requests a specific address, if that address isn't in the cache obtained in step 3, a new authorization request is made to see if the peers requested address is allowable. This step can be repeated multiple times until both sides agree on the remote peer's address or until the NAS (or client) decide they're never going to agree and they shut down PPP instead.

4.3.8.21 Authorization Relies on Authentication

Since we pretty much rely on having a username in authorization requests to decide which addresses etc. to hand out, it is important to know where the username for a PPP user comes from. There are generally 2 possible sources:

1. You force the user to authenticate by making her login to the exec and you use that login name in authorization requests. This username isn't propagated to PPP by default. To have this happen, you generally need to configure the `if-needed` method, e.g.

```
aaa authentication login default tacacs+
aaa authentication ppp default if-needed
```

2. Alternatively, you can run an authentication protocol, PAP or CHAP (CHAP is much preferred), to identify the user. You don't need an explicit login step if you do this (so it's the only possibility if you are using autoselect). This authentication gets done before you see the first LCP authorization request of course. Typically you configure this by doing:

```
aaa authentication ppp default tacacs+
int async 1
  ppp authentication chap
```

If you omit either of these authentication schemes, you will start to see authorization requests in which the username is missing.

4.3.8.22 Configuring Service Authorization

A list of AV pairs is placed in the daemon's configuration file in order to authorize services. 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 AV pairs before returning them to the NAS, thereby restricting what the user is permitted to do.

4.3.8.22.1 The Authorization Algorithm

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

Find the user (or group) entry for this service (and protocol), then for each AV pair sent from the NAS:

1. If the AV pair from the NAS is mandatory:
   (a) look for an exact attribute,value match in the user's mandatory list. If found, add the AV pair to the output.
   (b) If an exact match doesn't exist, look in the user'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.
2. If the AV pair from the NAS is optional:
   (a) look for an exact attribute,value match in the user's mandatory list. If found, add DAEMON's AV pair to output.
   (b) If not found, look for the first attribute match in the user's mandatory list. If found, add DAEMON's AV pair to output.
   (c) 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.
   (d) 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.
   (e) If no match is found, delete the AV pair if the default is deny, or
   (f) If the default is permit add the NAS AV pair to the output.
3. 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 (but add only ONE AV pair for each mandatory attribute).
4. After all AV pairs have been processed, for each optional unrequested DAEMON AV pair, if there is no attribute match already in the output list, add that AV pair (but add only ONE AV pair for each optional attribute).

4.3.8.22.2 Recursive Authorization

Remember that authorization is also recursive over groups. 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.

4.3.8.23 Examples

```
host = 0.0.0.0/0 {
    key = "your key here"
}

group = admin {
    # group members who have no expiry date set will use this one
    valid until = 1997-01-01
    service = shell {
        default cmd = permit
    }
}

user=fred {
    login = crypt mEX027bHtzTlQ
    member = admin
    valid until = 2005-05-23

    service = shell {
        # When Fred starts an exec, his connection access list is 5
        set acl = 5

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

        # All commands except show system are denied for Fred
        cmd = show {
            # Fred can run the following show command
            permit system
            deny .
        }
    }

    service = ppp {
        protocol = lcp
        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
            set addr=131.108.12.11
            set addr=131.108.12.12
            set addr=131.108.12.13
            set addr=131.108.12.14

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

            # We will suggest an output access list of 102,
            # but the NAS may choose to ignore or override it
            optional outacl=102
        }
    }

    service = slip {
```

```
        default protocol = permit
        # Fred can run SLIP. When he does, he will have to use
        # these mandatory access lists
        set inacl=101
        set outacl=102
    }
}

user = wilma {
    login = crypt mEX027bHtzTlQ
    member = admin
}
```

4.3.8.24 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 15 cmdlist group tacacs+ local
```

and, on the lines:

```
aaa authorization commands 1 cmdlist
aaa authorization commands 15 cmdlist
```

This will make the NAS send TACACS+ requests for all level 1 (ordinary user) and level 15 (privileged level) commands.

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 group 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 {
    default service = permit
    service = shell { default cmd = permit }
    login = crypt kppPfHq/j6gXs
}
```

5 MAVIS Backends

The distribution comes with various *MAVIS* modules, of which the *external* module is probably the most interesting, as it interacts with simple Perl scripts to authenticate and authorize requests. You'll find sample scripts in the `mavis/perl` directory. Have a close look at them, as you may (or will) need to perform some trivial customizations to make them match your local environment.

You should really have a look at the *MAVIS* documentation. It gives examples for RADIUS and PAM authentication, too.

5.1 LDAP Backends

`mavis_tacplus_ldap.pl` is an authentication/authorization backend for the *external* module. It interfaces to various kinds of LDAP servers, e.g. OpenLDAP, Fedora DS and Active Directory. Its behaviour is controlled by a list of environmental variables:

Variable	Description
`LDAP_SERVER_TYPE`	One of: `generic`, `tacacs_schema`, `microsoft`. Default: `tacacs_schema`
`LDAP_HOSTS`	Space-separated list of LDAP URLs or IP addresses or hostnames Examples: `"ldap01 ldap02"`, `"ldaps://ads01:636 ldaps://ads02:636"`
`LDAP_SCOPE`	LDAP search scope (`base`, `one`, `sub`) Default: `sub`
`LDAP_BASE`	Base DN of your LDAP server Example: `dc=example,dc=com`

`LDAP_FILTER`	LDAP search filter. Defaults:
	• for `LDAP_SERVER_TYPE=generic`:
	`"(uid=%s)"`
	• for `LDAP_SERVER_TYPE=tacacs_schema`:
	`"(&(uid=%s)(objectClass=tacacsAccount))"`
	• for `LDAP_SERVER_TYPE=microsoft`:
	`"(&(objectclass=user)(sAMAccountName=%s))"`

`LDAP_FILTER_CHPW`	LDAP search filter for password changes. Defaults:
	• for `LDAP_SERVER_TYPE=generic`:
	`"(uid=%s)"`
	• for `LDAP_SERVER_TYPE=tacacs_schema`:
	`"(&(uid=%s)(objectClass=tacacsAccount)(!(tacacsFlag=staticpas`
	• for `LDAP_SERVER_TYPE=microsoft`:
	`"(&(objectclass=user)(sAMAccountName=%s))"`

`LDAP_USER`	User to use for LDAP bind if server doesn't permit anonymous searches. Default: unset
`LDAP_PASSWD`	Password for `LDAP_USER` Default: unset
`AD_GROUP_PREFIX`	An AD group starting with this prefix will be used as the user's TACACS+ group membership. The value of `AD_GROUP_PREFIX` will be stripped from the group name. Example: With `AD_GROUP_PREFIX` set to `tacacs` (which is actually the default), an AD group membership of `TacacsNOC` will assign the user to the `NOC` TACACS+ group. Note that TACACS+ group names are case-sensitive.
`REQUIRE_AD_GROUP_PREFIX`	If set, user needs to be in one of the `AD_GROUP_PREFIX` groups. Default: unset
`USE_TLS`	If set, the server is required to support `start_tls`. Default: unset
`FLAG_CHPW`	Permit password changes via this backend. Default: unset
`FLAG_PWPOLICY`	Try to enforce a simplicistic password policy. Default: unset

Variable	Description
FLAG_CACHE_CONNECTION	Keep connection to LDAP server open. Default: unset
FLAG_FALLTHROUGH	If searching for the user in LDAP fails, try the next MAVIS module (if any). Default: unset
FLAG_USE_MEMBEROF	Use the memberOf attribute for determining group membership. Setting LDAP_SERVER_TYPE to microsoft implies this. May be used if you're running *OpenLDAP* with **memberof** overlay enabled. Default: unset

5.1.1 LDAP Custom Schema Backend

For LDAP_SERVER_TYPE set to tacacs_schema, the program expects the LDAP server to support the experimental ldap.schema included for OpenLDAP and Fedora-DS. The schema files are located in the mavis/perl directory.

The new schema allows for a *auxiliary* object class

```
objectClass: tacacsAccount
```

which introduces a couple of new attributes. A sample user entry could then look similar to the following LDIF snippet:

```
dn: uid=marc,ou=people,dc=example,dc=com
uid: marc
cn: Marc Huber
objectClass: posixAccount
objectClass: inetOrgPerson
objectClass: shadowAccount
objectClass: tacacsAccount
shadowMax: 10000
uidNumber: 1000
gecos: Marc Huber
givenName: Marc
sn: Huber
gidNumber: 500
shadowLastChange: 14012
loginShell: /bin/bash
homeDirectory: /Users/marc
mail: marc@example.com
userPassword:: abcdefghijklmnopqrstuvwxyz=
tacacsClient: 192.168.0.0/24
tacacsClient: management
tacacsMember: readonly@172.16.5.0/24
tacacsMember: readwrite@nasgroup
tacacsProfile: { valid until = 2010-01-30 chap = clear ahzoi5Ue }
```

As tacacsProfile may (and most probably will) contain sensitive data, you should consider setting up LDAP ACLs to restrict access.

You should be pretty familiar with OpenLDAP (or, for that matter, Fedora-DS) if you're willing to go this route. For current versions of OpenLDAP: Use ldapadd to add tacacs_schema.ldif to the cn=config tree. For older versions, add tacacs.schema to the list of included schema and objectClass definitions in slapd.conf.

5.1.2 Active Directory Backend

If LDAP_SERVER_TYPE is set to microsoft, the script backends to AD servers. Sample configuration:

```
id = spawnd {
    listen = {
        port = 49
```

```
    }
    spawn = {
        instances min = 1
        instances max = 10
    }
    background = yes
}

id = tac_plus {
    access log = /var/log/tacacs/%Y/%m/%d/access.log
    accounting log = /var/log/tacacs/%Y/%m/%d/acct.log

    mavis module = external {
        # Optionally:
        # script out = {
        #     # Require group membership:
        #     if (undef($TACMEMBER) && $RESULT == ACK) set $RESULT = NAK
        #
        #     # Don't cache passwords:
        #     if ($RESULT == ACK) set $PASSWORD_ONESHOT = 1
        # }

        setenv LDAP_SERVER_TYPE = "microsoft"
        # setenv LDAP_HOSTS = "ldaps://ads01:636 ldaps://ads02:636"
        setenv LDAP_HOSTS = "ads01:3268 ads02:3268"
        setenv LDAP_SCOPE = sub
        setenv LDAP_BASE = "dc=example,dc=com"
        setenv LDAP_FILTER = "(&(objectclass=user)(sAMAccountName=%s))";
        setenv LDAP_USER = tacacs@example.com
        setenv LDAP_PASSWD = Secret123
        setenv AD_GROUP_PREFIX = tacacs
        # setenv REQUIRE_AD_GROUP_PREFIX = 1
        setenv USE_TLS = 0
        exec = /usr/local/lib/mavis/mavis_tacplus_ldap.pl
    }

    login backend = mavis
    pap backend = mavis

    host = world {
        address = ::/0
        welcome banner = "Welcome\n"
        key = cisco
    }

    host = helpdesklab {
        address = 192.168.34.16/28
    }

    # A user will be in the "admin" group if he's member of the
    # corresponding "tacacsadmin" ADS group.

    group = admin {
        default service = permit
        service = shell {
            default command = permit
            default attribute = permit
            set priv-lvl = 15
        }
    }

    # A user will be in the "helpdesk" group if he's member of the
```

```
    # corresponding "tacacshelpdesk" ADS group:

    group = helpdesk {
        default service = permit
        service = shell {
            default command = permit
            default attribute = permit
            set priv-lvl = 1
        }
        enable = deny
        member = admin@helpdesklab
    }
}
```

5.1.3 Generic LDAP Backend

If `LDAP_SERVER_TYPE` is set to `generic`, the script won't require any modification to your LDAP server, but only authenticates users (with `login = mavis`, `pap = mavis` or `password = mavis` declaration) defined in the configuration file. No authorization is done by this backend.

5.2 PAM backend

Example configuration for using *Pluggable Authentication Modules*:

```
id = spawnd { listen = { port = 49 } }

id = tac_plus {
  mavis module = groups {
    resolve gids = yes
    groups filter = /^(guest|staff)$/
    script out = {
      # copy the already filtered UNIX group access list to TACMEMBER
      eval $GIDS =~ /^(.*)$/
      set $TACMEMBER = $1
    }
  }
  mavis module = external {
    exec = /usr/local/sbin/pammavis pammavis "-s" "sshd"
  }
  user backend = mavis
  login backend = mavis
  host = global { address = 0.0.0.0/0 key = cisco }

  group = staff {
    service = shell {
      default command = permit
      set priv-lvl = 15
    }
  }
  group = guest {
    service = shell {
      default command = deny
      set priv-lvl = 15
      cmd = show { permit .* }
    }
  }
}
```

5.3 System Password Backends

`mavis_tacplus_passwd.pl` authenticates against your local password database. Alas, to use this functionality, the script may have to run as root, as it needs access to the encrypted passwords. Primary and auxiliary UNIX group memberships will be mapped to TACACS+ groups.

`mavis_tacplus_opie.pl` is based on `mavis_tacplus_passwd.pl`, but uses OPIE one-time passwords for authentication.

```
id = spawnd {
  listen = { port = 49 }
}

id = tac_plus {
  mavis module = external {
    exec = /usr/local/lib/mavis/mavis_tacplus_opie.pl
  }
  login backend = mavis chalresp
  host = global { address = 0.0.0.0/0 key = cisco }
  group = staff {
    service = shell {
      default command = permit
      set priv-lvl = 15
    }
  }
  user = user1 { password = mavis member = staff }
  user = user2 { password = clear pass2 member = staff }
}
```

5.4 Shadow Backend

`mavis_tacplus_shadow.pl` may be used to keep user passwords out of the **tac_plus**configuration file, enabling users to change their passwords via the password change dialog. Passwords are stored in an auxiliary, `/etc/shadow`-like ASCII file, one user per line:

```
username:encryptedPassword:lastChange:minAge:maxAge:reserved
```

`lastChange` is the number of days since 1970-01-01 when the password was last changed, and `minAge` and `maxAge` determine whether the password may/may not/needs to be changed. Setting `lastChange` to `0` enforces a password change upon first login.

Example shadow file:

```
marc:$1$q5/vUEsR$jVwHmEw8zAmgkjMShLBg/.:15218:0:99999:
newuser:$1$pQtQsMuj$GKpIr5r2GNaZNfDfnCBtw.:0:0:99999:
test:$1$pQtQsMuj$GKpIr5r2GNaZNfDfnCBtw.:15218:1:30:
```

Sample daemon configuration:

```
...
  id = tac_plus {
    ...
    mavis module = external {
      setenv SHADOWFILE = /path/to/shadow
      # setenv FLAG_PWPOLICY=y
      # setenv ci=/usr/bin/ci
      exec = /usr/local/lib/mavis/mavis_tacplus_shadow.pl
    }
    ...
    login backend = mavis chpass
```

```
    ...
    user = marc {
      login = mavis
      ...
    }
  ...
  }
...
```

5.5 RADIUS Backends

`mavis_tacplus_radius.pl` authenticates against a RADIUS server. No authorization is done, unless the `RADIUS_GROUP_ATTR` environment variable is set (see below). This module may, for example, be useful if you have static user account definitions in the configuration file, but authentication passwords should be verified by RADIUS. Use the `login = mavis` or `password = mavis` statement in the user profile for this to work.

If the `Authen::Radius` Perl module is installed, the value of the RADIUS attribute specified by `RADIUS_GROUP_ATTR` will be used to create a `TAC_MEMBER` definition which uses the attribute value as group membership. E.g., an attribute value of `Administrator` would result in a

```
  member = Administrator
```

declaration for the authenticated user, enabling authorization and omitting the need for static users in the configuration file.

Keep in mind that authorization will only work well if either

- the `tacplus_info_cache` module is being used (it will cache authentication AV pairs locally, so subsequent authorizations should work fine unless you're switching to a tac_plus server running elsewhere).

or

- `single-connection` is used and
- `mavis cache timeout` is set to a sufficiently high value that covers the user's (expected) maximum login time.

Alternatively to `mavis_tacplus_radius.pl` the `pamradius` program may called by the `external` module. Results should be roughly equivalent.

5.5.1 Sample Configuration

```
## Use tacinfo_cache to cache authorization data to disk:
mavis module = tacinfo_cache {
    directory = /tmp/tacinfo
}

## You can use either the Perl module ...
#mavis module = external {
#   exec = /usr/local/lib/mavis_tacplus_radius.pl
#   setenv RADIUS_HOST = 1.2.3.4:1812 # could add more hosts here, comma-separated
#   setenv RADIUS_SECRET = "mysecret"
#   setenv RADIUS_GROUP_ATTR = Class
#   setenv RADIUS_PASSWORD_ATTR = Password # defaults to: User-Password
# }
## ... or the freeradius-client based code:
mavis module = external {
    exec = /usr/local/sbin/radmavis radmavis "group_attribute=Class" "authserver ↩
        =1.2.3.4:1812:mysecret"
}
```

5.6 Experimental Backends

`mavis_tacplus_sms.pl` is a sample (skeleton) script to send One-Time Passwords via a SMS backend.

5.7 Error Handling

If a backend script fails due to an external problem (e.g. LDAP server unavailability), your router may or may not fall back to local authentication (if configured). Chances are, that the fallback doesn't work. If you still want to be able to authenticate via TACACS+ in that case, you can do so with a non-MAVIS user which will only be valid in case of a backend error:

```
...
# set the time interval you want the user to be valid if the backend fails:
authentication fallback period = 60 # that's actually the default value
...
# add a local user for emergencies:
user = cisco {
    ...
    fallback-only
    ...
}
```

To indicate that fallback mode is actually active, you may a display a different login prompt to your users:

```
host = ... {
    ...
    welcome banner = "Welcome\n"
    welcome banner fallback = "Welcome\nEmergency accounts are currently enabled.\n"
    ...
}
```

Fallback can be enabled/disabled globally an on a per-host basis. Default is enabled.

```
authentication fallback = permit
host = ... {
    ...
    authentication fallback = deny
    ...
}
```

6 Debugging

6.1 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 config-file
```

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

6.2 Trace Options

Trace (or debugging) options may be specified in *global*, *host*, *user* and *group* context. The current debugging level is a combination (read: `OR`) of all those. Generic syntax is:

```
debug = option ...
```

For example, getting command authorization to work in a predictable way can be tricky - the exact attributes the NAS sends to the daemon may depend on the IOS version, and may in general not match your expectations. If your regular expressions don't work, add

```
debug = REGEX
```

where appropriate, and the daemon may log some useful information to `syslog`.

Multiple trace options may be specified. Example:

```
debug = REGEX CMD
```

Trace options may be removed by prefixing them with−. Example:

```
debug = ALL -PARSE
```

The debugging options available are summarized in the following table:

Bit	Value	Name	Description
0	1	PARSE	Configuration file parsing
1	2	AUTHOR	Authorization related
2	4	AUTHEN	Authentication related
3	8	ACCT	Accounting related
4	16	CONFIG	Configuration related
5	32	PACKET	Packet dump
6	64	HEX	Packet hex-dump
7	128	LOCK	File locking
8	256	REGEX	Regular expressions
9	512	ACL	Access Control Lists
10	1024	RADIUS	unused
11	2048	CMD	Command lookups
12	4096	BUFFER	Buffer handling
13	8192	PROC	Procedural traces
14	16384	NET	Network related
15	32768	PATH	File system path related
16	65536	CONTROL	Control connection related
17	131072	INDEX	Directory index related
18	262144	AV	Attribute-Value pair handling
19	524288	MAVIS	MAVIS related
20	1048576	DNS	DNS related
21	2097152	USERINPUT	Show user input (this may include passwords)
31	2147483648	NONE	Disable debugging

Some of those debugging options are not used and trigger no output at all.

Debugging User Input

The daemon will (starting with snapshot 202012051554) by default no longer outputs user input from authentication packets sent by the NAS. You can explicitly change this using the `USERINPUT` debug flag. Something like

```
debug = ALL
```

or using a numeric value will *not* work, it needs to be enabled explicitly, e.g.:

```
debug = ALL USERINPUT
```

Be prepared to see plain text user passwords if you enable this option.

---

7 Frequently Asked Questions

- **Is there a Graphical User Interface of any kind?**

  Old answer: No, unless your favourite text editor does qualify. The configuration syntax is too complex and flexible to be coverable using a GUI. If you're looking for a way to manage your user database then have a look at Webmin, ActiveDirectory, whatever. It's trivial to use those as authentication/authorization backends.

  Updated answer: Yes, various, and unrelated. Search for "tacacsgui". There are other adaptors, too.

- **How do I use the MySQL backend?**

  You don't, unless you're willing to write the SQL backend yourself. This isn't hard, really, but obviously depends a lot on your database structure and there's simply no generic enough way to free you from this step. Just have a look at, e.g., the LDAP Perl scripts that come with the distribution. If you're remotely familiar with Perl and your SQL database, then writing the backend will be a piece of cake.

- **I'm using the *single-connection* feature. How can I force my router to close the TCP connections to the TACACS+ server?**

  On IOS, `show tcp brief` will display the TCP connections. Search for the ones terminating at your server, and kill them using `clear tcp tcb ...`. Example:

```
Router#sho tcp brief | incl 10.0.0.1.49
633BB794  10.0.0.2.17326              10.0.0.1.49                ESTAB
6287E4C4  10.0.0.2.24880              10.0.0.1.49                ESTAB
Router#clear tcp tcb 633BB794
[confirm]
 [OK]
Router#clear tcp tcb 6287E4C4
[confirm]
 [OK]
Router#
```

- **Does the daemon require a working DNS?**

  No, DNS is not a requirement. You may however use DNS reverse mapping in NAC ACL lists, provided that the software is compiled with `c-ares` support.

- **Why do I get PPP authorization failures because of `no username in request` when I've already logged in and authenticated?**

  With `aaa authentication ppp default group tacacs+`, the ppp authentication overrides the earlier login authentication. If the ppp authentication fails, the username ends up blank. Changing the config to `aaa authentication ppp default if-needed group tacacs+` fixes this problem.

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

  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 late one night.

  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 extended 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+.

With the scheme choosen for newer TACACS+ protocol revisions, the NAS sends the challenge information to the TACACS+ daemon and the daemon uses the cleartext password to generate the response and returns that.

The original TACACS+ protocol included specific protocol knowledge for both ARAP and CHAP. Please note that this version of the daemon implementation no longer supports SENDPASS, SENDAUTH and ARAP to comply to RFC8907.

However, the above doesn't apply to PAP. You can keep an inbound PAP password DES- or MD5-encrypted, since all you need to do with it is verify that the password the principal gave you is correct.

- **How is the typical login authentication sequence done?**

  1. NAS sends START packet to daemon
  2. Daemon send GETUSER containing login prompt to NAS
  3. NAS prompts user for username
  4. NAS sends packet to daemon
  5. Daemon sends GETPASS containing password prompt to the NAS
  6. NAS prompts user for password
  7. NAS sends packet to daemon
  8. Daemon sends accept, reject or error to NAS

- **What does "`default service = permit`" really do?**

  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=shell`.

  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 commands, there must be a matching `cmd=`*cmdname* or a `default cmd = permit` definition.

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

- **How do I make PAP work?**

  Avoid using PAP if possible since it's not very secure. If you *must* use it, PAP passwords may be specified along with ARAP and CHAP passwords for each user.

  Passwords
  It is very bad practice to use an outbound PAP password that is the same as any of your inbound passwords. For this reason, a "global" password, defined with the `password = clear ...` statement, does not apply to outbound PAP, only to inbound PAP, bidirectional CHAP and ARAP.

- **How can I deny telneting from a commserver by IP address only i.e. when command is `10.0.1.6` rather than `telnet 10.0.1.6`?**

  The best way to restrict telnet access is by applying an outbound access list via the access class command (or equivalently, via the "acl" avpair). The NAS configuration command

```
access-class n out
```

  for example applies a pre-defined standard IP access list (where n is a number from 1 through 99) that governs telnet access from a NAS.

  E.g. the following configuration commands permit outgoing Telnet access from line 1 on the NAS *only* to hosts on network 192.85.55.0:

```
access-list 12 permit 192.85.55.0 0.0.0.255
line 1
  access-class 12 out
```

Note: You must define the access-list on the NAS. Only then can you use the acl avpair to apply it to a line that a user dials in on.

Alternatively, you can try configuring `transport preferred none` on the lines in question. This will force a user to always type `telnet 10.0.1.6` in order to telnet out from the NAS. Then you can apply command authorization to this command to restrict it.

- **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?

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.

- **Can TACACS+ only be enabled on a global basis?**

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
```

Note that unfortunately, you can't (yet) apply authorization differently to selected lines and interfaces.

- **I have leased lines running PPP, and AAA authorization is also configured, so the authorization on the leased lines fails. What should I do?**

Since you can't (yet) configure authorization on a per-line basis, you have to turn on authentication on the leased lines running PPP and configure your TACACS+ server so that it will authorize these lines correctly.

A more demanding alternative is to modify the TACACS+ server source code to allow any authorizations coming in from the port "SerialXXX" to succeed.

A third possibility is to not use PPP on those lines, e.g. use HDLC instead. HDLC doesn't require authentication or authorization.

- **How many characters may a TACACS+ Username and Password contain?**

The short answer is 31 bytes of username, with up to 254 bytes of password if they are cleartext (8 bytes if passwords are DES encrypted).

The long answer is that the Cisco NAS allocates a buffer of 1024 bytes, so this is the maximum you can type in, in response to a NAS prompt.

But the protocol spec allows a username or password length field of just one byte in an authentication packet, so only the first 255 of these characters can be sent to the daemon.

Now if it's a DES encrypted password, then only the first 8 bytes are significant, per the common unix implementations of crypt.

- **How do I limit the number of sessions a user can have?**

With this version of the daemon you can't.

- **How can I configure time-outs on an interface via TACACS+?**

  Certain per-user/per-interface timeouts may be set by TACACS+ during authorization. As of 11.0, you can set an arap session timeout, and an exec timeout. As of 11.1 you can also set an exec idle timeout.

  There are currently no settable timeouts for PPP or SLIP sessions, but there is a workaround which applies to ASYNC PPP/SLIP idle timeouts started via exec sessions only: This workaround is to set an EXEC (idletime) timeout on an exec session which is later used to start up PPP or SLIP (either via a TACACS+ autocommand or via the user explicitly invoking PPP or SLIP). In this case, the exec idle timeout will correctly terminate an idle PPP or SLIP session. Note that this workaround cannot be used for sessions which autoselect PPP or SLIP.

  An idle timeout terminates a connection when the interface is idle for a given period of time (this is equivalent to the "session-timeout" Cisco IOS configuration directive). The other timeouts are absolute. Of course, any timeouts set by TACACS+ apply only to the current connection.

```
user = lol {
    login = clear foobar
    service = shell {
        set idletime = 5 # disconnect lol if there is no traffic for 5 minutes
        set timeout = 60 # disconnect lol unconditionally after one hour
    }
}
```

  You also need to configure exec authorization on the NAS for the above timeouts, e.g.

```
aaa authorization exec default group tacacs+
```

  Note that these timeouts only work for async lines, not for ISDN currently.

  Note also that you cannot use the authorization `if-authenticated` option with these parameters, since that skips authorization if the user has successfully authenticated.

- **How do I send VPDN forwarding decisions to an authorization server?**

  In 11.2 onwards, VPDN NASs can use TACACS+ to allow an authorization server to make the decision to forward users.

  If VPDN forwarding is turned on, and the username is of the form `user@domain`, and `domain` matches a vpdn outgoing configured domain, then an authorization attempt is made for `domain` (see below).

  When making an authorization call for VPDN, a service type of `ppp` with a protocol type of `vpdn`, with a username of `domain` will be made e.g. when a PPP user comes up on a line with the username `foo@bar.com`, if `vpdn enable` and `aaa authorization ....` is enabled on the box, then a one-time authorization of the name `bar.com` is attempted.

  If this authorization is successful, no local authentication is attempted on the NAS, and the connection is forwarded via VPDN instead.

  If no VPDN-specific information comes back from this authorization call, the login proceeds as follows:

  If `tacacs-server directed-requests` are configured (note: this is true by default), then IOS will strip off the domain part of a name of the form `user@domain` and use "`domain`" to try and select a TACACS+ server. If successful, the username portion "`user`", without the domain, will be used for all subsequent authentication, authorization and accounting.

  If directed requests are turned off, then the entire username `user@domain` is treated as a username.

  vpdn specific information includes the attributes `tunnel-id`, `source-ip` (deprecated) and `ip-addresses`:

tunnel-id	This AV pair specifies the username that will be used to authenticate the tunnel over which the individual user MID will be projected. This is analogous to the "NAS name" in the "vpdn outgoing" command.
ip-addresses	This is a list of possible IP addresses that can be used for the end-point of the tunnel. Consult the text at the end of this document for more details on how to configure this attribute.

source-ip	*This is now deprecated. It began in release 11.2(1.4), and was removed in 11.2(4.0.2).* This ip address will be used as the source of all VPDN packets generated as part of the VPDN tunnel (see the source-ip keyword in the vpdn outgoing command).

TACACS+ syntax

The following syntax is used:

```
user = domain {
  service = ppp {
    protocol = vpdn {
      set tunnel-id = name for tunnel authentication
      set ip-addresses = addr [addr ...]
      set source-ip = ip-address
    }
  }
}
```

In addition the TACACS+ server can be used to store the usernames for both the NAS (the username specified by "tunnel-id" above) and the Home Gateway. These will be used to authenticate the tunnel.

Example:

```
user = foobar.example.com {
  service = ppp {
    protocol = vpdn {
      set tunnel-id = my_nas
      set ip-addresses = "203.0.113.19 203.0.113.20"
      set source-ip = 197.51.100.1
    }
  }
}

user = my_nas { chap = clear egad }

user = my_home_gateway { chap = clear wowser }
```

IOS Configuration

To enable AAA assisted VPDN forwarding on a NAS, the following minimum configuration is required:

```
vpdn enable
aaa new-model
aaa authorization network default group tacacs+ ...
```

In addition, if the passwords for the home gateway and NAS are stored on the TACACS+ daemon, the command

```
aaa authentication login default group tacacs+ ....
```

should also be present.

Beginning with release 11.2(1.4), the additional configuration

```
vpdn outgoing example.com ip NAS [ source-ip X.X.X.X ]
```

can be used. This changes in 11.2(4.0.2) and becomes:

```
vpdn source-ip X.X.X.X
vpdn outgoing example.com ip NAS
```

- **Can someone expand on the use of the `optional` keyword?**

  Most attributes are mandatory i.e. if the daemon sends them to the NAS, the NAS must obey them or deny the authorization. This is the default. It is possible to mark attributes as optional, in which case a NAS which cannot support the attribute is free to simply ignore it without causing the authorization to fail.

  This was intended to be useful in cutover situations where you have multiple NASes running different versions of IOS, some of which support more attributes than others. If you make the new attributes optional, older NASes could ignore the optional attributes while new NASes could apply them. Note that this weakens your security a little, since you are no longer guaranteed that attributes are always applied on successful authorization, so it should be used judiciously.

- **Can I do do address pooling on the TACACS+ daemon?**

  Not really since nothing on the daemon tracks whether an address is already in use. The best way to manage address pools is to define a non-overlapping pool of addresses on each NAS and return the name of this pool during authorization whenever a user logs in e.g.

  NAS:

```
ip local pool foo 1.0.0.1 1.0.0.10
```

  Daemon:

```
user = lol { service = ppp { protocol = ip { set addr-pool = foo } } }
```

- **What about MSCHAP?**

  The daemon contains mschap support. Mschap is configured the same way as chap, only using the `mschap` keyword in place of the `chap` keyword.

  Like ARAP, MSCHAP requires DES support. Use the `--with-ssl` flag when configuring the package.

8 Canned Configurations

Here are some canned configurations for getting demos started:

8.1 Login Authentication

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:

```
id = spawnd {
    listen = { port = 49 }
}
id = tac_plus {
    host = any { key = "some key" address = 0.0.0.0/0 }

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

NAS:

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

8.2 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:

```
id = spawnd {
    listen = { port = 49 }
}
id = tac_plus {
    host = any { key = "some key" address = 0.0.0.0/0 }

    # repeat as necessary for each user
    user = fred {
        login = clear abcdef
        service = shell {
            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 default group tacacs+ none
tacacs-server host 10.1.1.1
tacacs-server key some key
```

8.3 Network Access Authorization

This config allows `fred` to login to `line 1` with password `abcdef` (or to and to run ppp using chap authentication. The chap password is `lab`.

Daemon:

```
id = spawnd {
    listen = { port = 49 }
}
id = tac_plus {
    host = any { key = "some key" address = 0.0.0.0/0 }

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

NAS:

```
aaa new-model
! authenticate exec logins (if not autoselecting)
aaa authentication login default group tacacs+
! authorize network services via TACACS+
aaa authorization network default group tacacs+
```

```
! use TACACS+ for authenticating ppp users
aaa authentication ppp default group tacacs+
tacacs-server host 10.1.1.1
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
```

8.4 ARAP

Daemon:

```
id = spawnd {
    listen = { port = 49 }
}
id = tac_plus {
    host = any { key = "some key" address = 0.0.0.0/0 }

    user = lol {
        arap = clear arapSecret
        service = arap
    }
}
```

NAS:

```
aaa new-model
aaa authentication arap default group tacacs+
aaa authorization network default group tacacs+
aaa accounting network default start-stop group tacacs+
!
appletalk routing
arap network ...
!
interface Ethernet0
  appletalk cable-range ...
  appletalk zone ...
!
tacacs-server host ...
tacacs-server key ...
!
line 1
  location a modem
  modem answer-timeout 0
  modem InOut
  autoselect arap
  autoselect during-login
  arap enable
  speed ...
  flowcontrol hardware
```

8.5 Callback

Availability of Callback
Callback is available only in IOS 11.1 and later, and can only be controlled via TACACS+ for ASYNC lines. ISDN callback can be configured on the NAS but cannot be controlled via AAA.

Here is an example of AAA configuration (with exec and network accounting enabled):

Daemon:

Example of remote TACACS+ server configuration file entry for username `foobar`:

```
id = spawnd {
    listen = { port = 49 }
}
id = tac_plus {
    host = any { key = "some key" address = 0.0.0.0/0 }

    user = foobar {
        arap = clear AAAA
        login = clear LLLL
        chap = clear CCCC
        pap = clear PPPP
        service = ppp {
            default protocol = permit
            protocol = lcp {
                set callback-dialstring=123456
            }
        }
        service = arap {
            protocol = atalk {
                set callback-dialstring=2345678
            }
        }
        service = shell {
            set callback-dialstring=3456789
            set callback-line=7
            set nocallback-verify=1
        }
    }
}
```

NAS:

```
aaa new-model
tacacs-server host XX.XX.XX.XX
tacacs-server key fookey
aaa accounting exec wait-start tacacs+
aaa accounting network wait-start tacacs+

! Example of AAA configuration for Exec:
aaa authentication login execcheck tacacs+
aaa authorization network tacacs+
service exec-callback

line 4
  login authentication execcheck

! Example of AAA configuration for ARAP:
aaa authentication arap arapcheck tacacs+
aaa authorization network tacacs+
```

```
arap callback

line 4
  arap authentication arapcheck

! Example of AAA-specific configuration for PPP callback:
aaa new-model
aaa authentication ppp pppcheck tacacs+
aaa authorization network tacacs+

int async 6
  ppp authentication chap pppcheck
  ppp callback accept
```

9 Authorization AV pairs

The following authorization AV pairs specify which service is being authorized and are typically accompanied by protocol AV pairs and other, additional pairs from the lists below. (For a complete and current list of supported AV pairs and required IOS releases search http://www.cisco.com/ for "TACACS+ Attribute-Value Pairs".)

- `service=arap`
- `service=shell`

  Used for exec startup, and also for command authorizations.
- `service=ppp`
- `service=slip`
- `service=system`

  Not used
- `service=raccess`

  Used for managing reverse telnet connections, e.g.:

```
user = jim {
login = clear lab
    service = raccess {
        set port#1 = nasname1/tty2
        set port#2 = nasname2/tty5
    }
}
```

  Requires IOS configuration:

```
aaa authorization reverse-access default group tacacs+
```

  See the IOS docs for more details.
- `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`

  Used with `service=ppp` or `service=arap`.

- `protocol=vines`

  For VINES over PPP.

- `protocol=ccp`

  Authorization of CCP (Compression Control Protocol). No other av-pairs associated with this.

- `protocol=cdp`

  Authorization of CDP (Cisco Discovery Protocol). No other av-pairs associated with this.

- `protocol=multilink`

  Authorization of multilink PPP. See `max-links` and `load-threshold`.

- `protocol=unknown`

  For undefined/unsupported conditions. Should not occur under normal circumstances.

- `cmd` (EXEC)

  If the value of cmd is NULL e.g. the AV pair is `cmd=`, then this is an authorization request for starting an exec (shell). 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` (EXEC)

  During command authorization, the name of the command is given by an accompanying `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` (ARAP, EXEC)

  For ARAP this contains an access-list number. For EXEC authorization it contains an access-class number, e.g. `acl=2`. which is applied to the line as the output access class equivalent to the configuration command

```
line ...
  access-class 2 out
```

  An outbound access-class is the best way to restrict outgoing telnet connections. Note that a suitable access list (in this case, numbered 2) must be predefined on the NAS.

- `inacl` (PPP/IP/IPX)

  This AV pair contains an IP or IPX input access list number for slip or PPP e.g. `inacl=2`. The access list itself must be pre-configured on the Cisco box. Per-user access lists do not work with ISDN interfaces unless you also configure a virtual interface. After 11.2(5.1)F, you can also use the name of a predefined named access list, instead of a number, for the value of this attribute. *Note:* For IPX, `inacl` is only valid after 11.2(4)F.

- `inacl#`*n* (PPP/IP, PPP/IPX, 11.2(4)F)

  This AV pair contains the definition of an input access list to be installed and applied to an interface for the duration of the current connection, e.g.

```
inacl#1="permit ip any any precedence immediate"
inacl#2="deny igrp 0.0.1.2 255.255.0.0 any"
```

  Attributes are sorted numerically before they are applied. For IP, standard OR extended access list syntax may be used, but it is an error to mix the two within a given access-list. For IPX, only extended access list syntax may be used. See also:

```
sho ip access-lists
sho ip interface
sho ipx access-lists
sho ipx interface
debug aaa author
debug aaa per-user
```

- `outacl` (PPP/IP, PPP/IPX)

  This AV pair contains an IP or IPX output access list number for SLIP. PPP/IP or PPP/IPX connections e.g. outacl=4. The access list itself must be pre-configured on the Cisco box. Per-user access lists do not work with ISDN interfaces unless you also configure a virtual interface. PPP/IPX is supported in 11.1 onwards only. After 11.2(5.1)F, you can also use the name of a predefined named access list, as well as a number, for the value of this attribute.

- `outacl#`*n* (PPP/IP, PPP/IPX, 11.2(4)F)

  This AV pair contains an output access list definition to be installed and applied to an interface for the duration of the current connection, e.g.

```
outacl#1="permit ip any any precedence immediate"
outacl#2="deny igrp 0.0.9.10 255.255.0.0 any"
```

  Attributes are sorted numerically before they are applied. For IP, standard OR extended access list syntax may be used, but it is an error to mix the two within a given access-list. For IPX, only extended access list syntax may be used.

  See also:

```
sho ip access-lists
sho ip interface
sho ipx access-lists
sho ipx interface
debug aaa author
debug aaa per-user
```

- `addr` (SLIP, PPP/IP)

  The IP address the remote host should be assigned when a slip or PPP/IP connection is made e.g. `addr=1.2.3.4`.

- `routing` (SLIP, PPP/IP)

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

- `timeout` (11.0 onwards, ARAP, EXEC)

  Sets the time until an arap or exec session disconnects unconditionally (in minutes), e.g. `timeout=60`.

- `autocmd` (EXEC)

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

- `noescape` (EXEC)

  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` (EXEC)

  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` (EXEC)

  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` (ARAP)

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

- `addr-pool` (11.0 onwards, PPP/IP, SLIP)

  This AV pair specifies the name of a local pool from which to get the IP address of the remote host. Note: `addr-pool` works in conjunction with local pooling. It specifies the name of a local pool (which needs to be pre-configured on the NAS). Use the ip-local pool command to declare local pools, e.g on the NAS:

```
ip address-pool local ip local pool foo 1.0.0.1 1.0.0.10
ip local pool baz 2.0.0.1 2.0.0.20
```

Then you can use TACACS+ to return `addr-pool=foo` or `addr-pool=baz` to indicate which address pool you want to get this remote node's address from, e.g. on the daemon:

```
user = lol { service = ppp { protocol = ip { set addr-pool=foo } }
```

- `route` (11.1 onwards, PPP/IP, SLIP).

  This AV pair specifies a route to be applied to an interface. During network authorization, the "route" attribute may be used to specify a per-user static route, to be installed via TACACS+. The daemon side declaration is:

  ```
  service = ppp {protocol=ip{set route="dst_addr mask [ gateway ]"}}
  ```

  This indicates a temporary static route that is to be applied. *dst_address*, *mask* and *gateway* are expected to be in the usual dotted-decimal notation, with meanings the same as for the familiar `ip route` configuration command on a NAS. If gateway is omitted, the peer's address is taken to be the gateway. The route is expunged once the connection terminates.

- `route#`*n* (PPP/IP/IPX, 11.2(4)F)

  Same as the `route` attribute, except that these are valid for IPX as well as IP, and they are numbered, allowing multiple routes to be applied e.g.

  ```
  route#1="3.0.0.0 255.0.0.0 1.2.3.4"
  route#2="4.0.0.0 255.0.0.0"
  ```

  or, for IPX,

  ```
  route#1="4C000000 ff000000 30.12.3.4"
  route#2="5C000000 ff000000 30.12.3.5"
  ```

  See also:

  ```
  sho ip route
  sho ipx route
  debug aaa author
  debug aaa per-user
  ```

- `callback-rotary` (11.1 onwards, valid for ARAP, EXEC, SLIP or PPP)

  The number of a rotary group (between 0 and 100 inclusive) to use for callback e.g. `callback-rotary=34`. Not valid for ISDN.

- `callback-dialstring` (11.1 onwards, valid for ARAP, EXEC, SLIP or PPP)

  sets the telephone number for a callback e.g. `callback-dialstring=408-555-1212`. Not valid for ISDN.

- `callback-line` (11.1 onwards, valid for ARAP, EXEC, SLIP or PPP)

  The number of a tty line to use for callback e.g. `callback-line=4`. Not valid for ISDN.

- `nocallback-verify` (11.1 onwards, valid for ARAP, EXEC)

  Indicates that no callback verification is required. The only valid value for this parameter is the digit one i.e. `nocallback-verify=` Not valid for ISDN.

- `idletime` (11.1 onwards, EXEC)

  Sets a value, in minutes, after which an IDLE session will be terminated. *Note:* Does NOT work for PPP.

- `tunnel-id` (11.2 onwards, PPP/VPDN)

  This AV pair specifies the username that will be used to authenticate the tunnel over which the individual user MID will be projected. This is analogous to the "NAS name" in the `vpdn outgoing` command.

- `ip-addresses` (11.2 onwards, PPP/VPDN)

  This is a space separated list of possible IP addresses that can be used for the end-point of the tunnel. In 11.2(5.4)F, this attribute was extended as follows:

  1. comma (','') is also considered as a delimiter. For example the avpair can now be written as

     ```
     ip-addresses = 172.21.9.26,172.21.9.15,172.21.9.4
     ```

  2. slash ('/') is considered a priority delimiter. When you have a number of Home Gateway routers, it is desirable to consider some as the primary routers and some as backup routers. '/' allow you to config the routers into priority groups, so that the NAS will try to forward the users to the high priority routers, before forwarding to the low priority one.

     For example in the following av-pair:

     ```
     ip-addresses = "172.21.9.26 / 172.21.9.15 / 172.21.9.4"
     ```

     172.21.9.26 is considered to be priority 1,

     172.21.9.15 is considered to be priority 2,

     172.21.9.4 is considered to be priority 3.

     The NAS will try to forward the users to 172.21.9.26, before trying 172.21.9.15. If the NAS can't forward users to 172.21.9.26, it will try 172.21.9.15 next. If it fails with 172.21.9.15, it will then try forwarding to 172.21.9.4.

- `source-ip` (PPP/VPDN, now deprecated, only existed in releases 11.2(1.4) thru 11.2(4.0.2))

  This specifies a single ip address will be used as the source of all VPDN packets generated as part of the VPDN tunnel (see the equivalent source-ip keyword in the IOS vpdn outgoing command).

- `nas-password` (PPP/VPDN, 11.2(3.4)F, 11.2(4.0.2)F)

  During L2F tunnel authentication, nas-password specifies the password for the NAS.

- `gw-password` (PPP/VPDN, 11.2(3.4)F, 11.2(4.0.2)F)

  During L2F tunnel authentication, gw-password specifies the password for the home gateway.

- `rte-ftr-in#`*n* (PPP -- IP/IPX, 11.2(4)F)

  This AV pair specifies an input access list definition to be installed and applied to routing updates on the current interface, for the duration of the current connection. For IP, both standard and extended Cisco access list syntax is recognised, but it is an error to mix the two within a given access-list. For IPX, only Cisco extended access list syntax is legal. Attributes are sorted numerically before being applied. For IP, the first attribute must contain the name of a routing process and its identifier (except for rip, where no identifier is needed), e.g.

  ```
  rte-fltr-in#0="router igrp 60"
  rte-fltr-in#1="permit 0.0.3.4 255.255.0.0"
  rte-fltr-in#2="deny any"
  ```

  For IPX, no routing process is needed, e.g.

  ```
  rte-fltr-in#1="deny 3C01.0000.0000.0001"
  rte-fltr-in#2="deny 4C01.0000.0000.0002"
  ```

  See also:

  ```
  show ip access-lists
  show ip protocols
  sho ipx access-lists
  sho ipx interface
  debug aaa author
  debug aaa per-user

  router ...
  [no] distribute-list ...

  ipx input-network-filter ...
  ```

- `rte-ftr-out#n` (PPP/IP, 11.2(4)F)

  This AV pair specifies an input access list definition to be installed and applied to routing updates on the current interface, for the duration of the current connection. For IP, both standard and extended Cisco access list syntax is recognised, but it is an error to mix the two within a given access-list. Attributes are sorted numerically before being applied. The first attribute must contain the name of a routing process and its identifier (except for rip, where no identifier is needed), e.g.

```
rte-fltr-out#0="router igrp 60"
rte-fltr-out#3="permit 0.0.5.6 255.255.0.0"
rte-fltr-out#4="permit any"
```

  For IPX, no routing process is specified, e.g.

```
rte-fltr-out#1="deny 3C01.0000.0000.0001"
rte-fltr-out#2="deny 4C01.0000.0000.0002"
```

  See also:

```
sho ipx access-lists
sho ipx interface
show ip access-lists
show ip protocols
debug aaa author
debug aaa per-user

router ...
  [no] distribute-list ...

ipx output-network-filter ...
```

- `sap#n` (11.2(4)F PPP/IPX)

  This AV pair specifies static saps to be installed for the duration of a connection e.g.

```
sap#1="4 CE1-LAB 1234.0000.0000.0001 451 4"
sap#2="5 CE3-LAB 2345.0000.0000.0001 452 5"
```

  The syntax of static saps is the same as that used by the IOS `ipx sap` command.

  See also:

```
sho ipx servers
debug aaa author
debug aaa per-user
[no] ipx sap ...
```

- `route#n` (PPP/IP/IPX, 11.2(4)F)

  Same as the `route` attribute, except that these are valid for IPX as well as IP, and they are numbered, allowing multiple routes to be applied e.g.

```
route#1="3.0.0.0 255.0.0.0 1.2.3.4"
route#2="4.0.0.0 255.0.0.0"
```

  or, for IPX,

```
route#1="4C000000 ff000000 30.12.3.4"
route#2="5C000000 ff000000 30.12.3.5"
```

  See also:

```
sho ip route
sho ipx route
debug aaa author
debug aaa per-user
```

- `sap-fltr-in#`*n* (PPP/IPX, 11.2(4)F)

  This AV pair specifies an input sap filter access list definition to be installed and applied on the current interface, for the duration of the current connection. Only Cisco extended access list syntax is legal, e.g

```
sap-fltr-in#1="deny 6C01.0000.0000.0001"
sap-fltr-in#2="permit -1"
```

  Attributes are sorted numerically before being applied.

  See also:

```
sho ipx access-lists
sho ipx interface
debug aaa author
debug aaa per-user
ipx input-sap-filter ...
```

- `sap-fltr-out#`*n* (PPP/IPX 11.2(4)F)

  This AV pair specifies an output sap filter access list definition to be installed and applied on the current interface, for the duration of the current connection. Only Cisco extended access list syntax is legal, e.g

```
sap-fltr-out#1="deny 6C01.0000.0000.0001"
sap-fltr-out#2="permit -1"
```

  Attributes are sorted numerically before being applied.

  See also:

```
sho ipx access-lists
sho ipx interface
debug aaa author
debug aaa per-user
ipx output-sap-filter ...
```

- `pool-def#`*n* (PPP/IP, 11.2(4)F)

  This attribute is used to define ip address pools on the NAS. During IPCP address negotiation, if an ip pool name is specified for a user (see the addr-pool attribute), a check is made to see if the named pool is defined on the NAS. If it is, the pool is consulted for an ip address. If the required pool is not present on the NAS (either in the local config, or as a result of a previous download operation), then an authorization call to obtain it is attempted, using the special username:

```
nas-name-pools
```

  where *nas-name* is the configured name of the NAS. *Note:* This username can be changed using the IOS configuration directive e.g.

```
aaa configuration config-name nas1-pools-definition.cisco.us
```

  The `pool-def` attribute is used to define ip address pools for the above authorization call e.g.

```
user = foo {
    login = clear lab
    service = ppp { protocol = ip { set addr-pool=bbb } }
}

user = nas1-pools {
    service = ppp {
        protocol = ip {
            set pool-def#1 = "aaa 1.0.0.1 1.0.0.3"
                set pool-def#2 = "bbb 2.0.0.1 2.0.0.10"
                set pool-def#3 = "ccc 3.0.0.1 3.0.0.20"
                set pool-timeout = 60
        }
    }
}
```

In the example above is a configuration file fragment for defining 3 pools named `aaa`, `bbb` and `ccc` on the NAS named `nas1`. When the user `foo` refers to the pool named `bbb`, if the pool `bbb` isn't defined, the NAS will attempt to download the definition contained in the `nas1-pools` entry. The other pools will also be defined at the same time (or they will be ignored if they are already defined). Since this procedure is only activated when an undefined pool is referenced, one way to redefine a pool once it has been downloaded is to manually delete the definition e.g. by logging into the NAS, enabling, and configuring:

```
config t
no ip local pool bbb
^Z
```

When a pool is deleted, there is no interruption in service for any user who is currently using a pool address. If a pool is deleted and then subsequently redefined to include a pool address that was previously allocated, the new pool will pick up the allocated address and track it as expected. Since downloaded pools do not appear in the NAS configuration, any downloaded pool definitions automatically disappear whenever a NAS reboots. These pools are marked as "dynamic" when they appear in the output of the `show ip local pools` NAS command. Since it is desirable not to have to manually delete pools to redefine them, the AV pair `pool-timeout=`*n* can be used to timeout any downloaded pool definitions. The timeout *n* is in minutes. The effect of the `pool-timeout` attribute is to start a timer when the pool definitions are downloaded. When the timer expires, the pools are deleted. The next reference to a deleted pool via will cause a re-fetch of the pool definitions. This allows pool changes to be made on the daemon and propagated to the NAS in a timely manner. See also:

```
sho ip local pool ...
ip local pool ...
```

- `old-prompts` (PPP/SLIP)

  This attribute allows providers to make the prompts in TACACS+ appear identical to those of earlier systems (TACACS and XTACACS). This will allow administrators to upgrade from TACACS/XTACACS to TACACS+ transparently to users. The difference between the prompts is as follows: In XTACACS, when the user types `slip` or `ppp` the system prompts for an address followed by a password, whereas TACACS+ prompts only for an address. In XTACACS, if the user types `slip host` or `ppp host`, the system prompts for a password. In TACACS+, there is no prompt. Using this attribute, TACACS+ can be made to mimic the prompting behaviour of xtacacs, by configuring network authorization on IOS, and using the `old-prompts=true` attribute value pair for slip and ppp/ip, viz:

```
user = joe {
    service = shell { }
    service = slip {
        default attribute = permit
        set old-prompts=true
    }
    service = ppp {
        protocol = ip {
            default attribute = permit
            set old-prompts=true
        }
    }
}
```

  i.e. the prompts are controlled by the addition of the `old-prompts=true` attribute.

- `max-links` (PPP/multilink - Multilink parameter; 11.3)

  This AV pair restricts the number of multilink bundle links that a user can have. The daemon side declaration is:

```
service=ppp { protocol=multilink { set max-links=n } }
```

  The range of *n* is [1-255].

  Related NAS commands:

```
int foo
  [no] ppp multilink

int virtual-template X
```

```
  multilink max-links n
show ppp multilink
debug multilink
```

- `load-threshold` (PPP/multilink - Multilink parameter; 11.3)

  This AV pair sets the load threshold at which an additional multilink link is added to the bundle (if load goes above) or deleted (if load goes below). The daemon side declaration is:

```
service=ppp { protocol=multilink { set load-threshold=n } }
```

  The range of *n* is [1-255]. Related NAS commands:

```
int foo
 [no] ppp multilink

int virtual-template X
  multilink load-threshold n

show ppp multilink

debug multilink
```

- `ppp-vj-slot-compression`

  Reserved for future use.

- `link-compression`

  Reserved for future use.

- `asyncmap`

  Reserved for future use.

- `x25-addresses` (PPP/VPDN)

  Reserved for future use.

- `frame-relay` (PPP/VPDN)

  Reserved for future use.

10 Upgrading from Previous Releases

There may be some caveats to consider if you're using a previous version of the software:

- 16-08-2009: To use the MAVIS backend for users not defined in the local configuration file, `user backend = mavis` is now required.
- 13-06-2010: LDAP functionality was merged into a single script, requiring the environmental variable `LDAP_SERVER_TYPE` to be set to `generic` if you were using the `mavis_tacplus_ldap_authonly.pl` script, to `tacplus_schema` for functionality equivalent to `mavis_tacplus_ldap.pl`, and to `microsoft` for `mavis_tacplus_ads.pl` functionality.

11 Bugs

- There may still be some nasty bugs lurking in the code. Please contact the author via the "Event-Driven Servers" Google Group at event-driven-servers@googlegroups.com or http://groups.google.com/group/event-driven-servers if you think you've found one.

- This documentation isn't well structured.
- The examples given are too IPv4-centric. However, the daemon handles IPv6 just fine.
- Some of the NAS configuration examples aren't recently tested. Refer to the IOS documentation for IOS configuration syntax guidance.

12 References

- draft-grant-tacacs-02.txt - The TACACS+ Protocol (Version 1.78)
- RFC8907: The Terminal Access Controller Access-Control System Plus (TACACS+) Protocol

13 Copyrights and Acknowledgements

Please see the source for copyright and licensing information of individual files.

- **The following applies if the software was compiled with OpenSSL support:**

  This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

  This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

- **MD4 algorithm:**

  The software uses the RSA Data Security, Inc. MD4 Message-Digest Algorithm.

- **MD5 algorithm:**

  The software uses the RSA Data Security, Inc. MD5 Message-Digest Algorithm.

- **If the software was compiled with PCRE (Perl Compatible Regular Expressions) support, the following applies:**

  Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England. (ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/).

- **The original tac_plus code (which this software and considerable parts of the documentation are based on) is distributed under the following license:**

  Copyright (c) 1995-1998 by Cisco systems, Inc.

  Permission to use, copy, modify, and distribute this software for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear on all copies of the software and supporting documentation, the name of Cisco Systems, Inc. not be used in advertising or publicity pertaining to distribution of the program without specific prior permission, and notice be given in supporting documentation that modification, copying and distribution is by permission of Cisco Systems, Inc.

  Cisco Systems, Inc. makes no representations about the suitability of this software for any purpose. THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

- **The code written by Marc Huber is distributed under the following license:**

  Copyright (C) 1999-2022 Marc Huber (Marc.Huber@web.de). All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: