TACACS+ NG

Marc Huber

COLLABORATORS			
	TITLE : TACACS+ NG		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Marc Huber	November 21, 2026	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction **1**

1.1 Download . . . 1

2 Definitions and Terms **1**

3 Operation **2**

3.1 Command line syntax . . . 2

3.2 Signals . . . 2

3.3 Event mechanism selection . . . 2

4 Configuration **3**

4.1 Sample Configuration . . . 3

4.2 Configuration directives . . . 7

4.2.1 Global options . . . 7

4.2.1.1 Limits and timeouts . . . 8

4.2.1.2 DNS . . . 8

4.2.1.3 Process-specific options . . . 9

4.2.1.4 Railroad Diagrams . . . 9

4.3 Realms . . . 9

4.3.1 Railroad Diagrams . . . 9

4.4 Realm attributes . . . 10

4.5 Logging . . . 10

4.5.1 Accounting . . . 14

4.5.2 Spoofing Syslog Packets . . . 15

4.6 User Messages . . . 15

4.7 Limits and timeouts . . . 16

4.7.1 Authentication . . . 16

4.7.2 User back-end options . . . 19

4.7.3 TLS and DTLS . . . 20

4.8 Miscellaneous . . . 22

4.9 Realm Inheritance . . . 23

4.10 Railroad Diagrams . . . 25

4.11 Networks . . . 26

4.11.1 Railroad Diagrams . . . 26

4.12 Devices (Hosts) . . . 27

4.12.1 Timeouts . . . 29

4.12.2 Authentication . . . 29

4.12.3 Authorization . . . 30

4.12.4 Banners and Messages . . . 31
4.12.5 Workarounds for Client Bugs . . . 31
4.12.6 Inheritance and Devices . . . 32
4.12.7 Railroad Diagrams . . . 32
4.12.8 Example . . . 33
4.13 Time Ranges . . . 34
4.13.1 Railroad Diagrams . . . 34
4.14 Access Control Lists . . . 34
4.14.1 Syntax . . . 36
4.15 Rewriting User Names . . . 42
4.16 Users . . . 42
4.16.1 Railroad Diagrams . . . 44
4.17 Groups . . . 45
4.17.1 Railroad Diagrams . . . 45
4.18 Profiles . . . 46
4.19 Railroad Diagrams . . . 48
4.20 Configuring Non-local Users via MAVIS . . . 51
4.21 Configuring Local Users for MAVIS authentication . . . 51
4.22 Configuring User Authentication . . . 51
4.23 Configuring Expiry Dates . . . 52
4.24 Configuring Authentication on the NAS . . . 52
4.25 Configuring Authorization . . . 53
4.26 Authorizing Commands . . . 53
4.27 The Authorization Process . . . 53
4.28 Authorization Relies on Authentication . . . 54
4.29 Configuring Service Authorization . . . 54
4.29.1 The Authorization Algorithm . . . 54

5 MAVIS Backends 55

5.1 LDAP Backends . . . 55
5.1.1 Multi-threaded LDAP Backend . . . 56
5.2 PAM back-end . . . 56
5.3 System Password Backends . . . 57
5.4 Shadow Backend . . . 57
5.5 RADIUS Backends . . . 58
5.5.1 Sample Configuration . . . 58
5.6 Experimental Backends . . . 58
5.7 Error Handling . . . 59

6 RADIUS **59**

6.1 Accepting RADIUS Queries . . . 60

6.2 RADIUS DACLs . . . 60

6.3 RADIUS Dictionary . . . 60

6.4 RADIUS Rule Set . . . 62

6.5 RADIUS Logging . . . 62

6.6 RADIUS Example Configurations . . . 62

7 Debugging **62**

7.1 Debugging Configuration Files . . . 62

7.2 Trace Options . . . 63

8 Frequently Asked Questions **64**

9 Multi-tenant setups **66**

9.1 AD, Realms and Tenants . . . 66

10 AAA rule tracing **67**

11 Bugs **69**

12 References **69**

13 Copyrights and Acknowledgements **69**

1 Introduction

tac_plus-ng is a TACACS+ daemon that supports RADIUS, too. It provides networking components like routers and switches with authentication, authorisation and accounting services.

This version is a major rewrite of the original public Cisco source code and is in turn largely based on **tac_plus**, which comes with the same distribution. Key features include:

- NAS specific device keys, prompts, enable passwords
- Rule-based permission assignment
- Flexible external back-ends 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.
- Full support for both IPv4 and IPv6
- Implements and auto-detects HAProxy protocol 2.
- Supports TACACS+ over TLS1.3
- Compliant to RFC8907
- Supports Linux VRFs
- Supports (non-standard) SSH Public Key Authentication (see the Wiki for reference)
- Implements and auto-detects legacy RADIUS (UDP and TCP), RADSEC (TLS) and RADIUS/DTLS (with PAP, CHAP, MSCHAPv1 and MSCHAPv2 authentication).
- Supports RADIUS Downloadable ACLs.

1.1 Download

You can download the source code from the GitHub repository at https://github.com/tacplus-ng/event-driven-servers/. On-line documentation is available via https://tacplus-ng.github.io/event-driven-servers/doc/index.html, too.

2 Definitions and Terms

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

Client or NAC	A Network Access Client, e.g. the source device of a `ssh` (or `telnet`) connection.
Device or NAS or NAD	A Network Access Server or Device, e.g. a Cisco box, or any other client which makes TACACS+ or RADIUS requests or generates 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*.

3 Operation

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

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-ng may use auxiliary **MAVIS** back-end modules for authentication of users and authorization of users and hosts.

3.1 Command line syntax

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

```
tac_plus-ng [ -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.

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-ng** 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
```

4 Configuration

The daemon is configured using a text file. Let's have a look at a sample configuration first, before digging into the various configuration directives.

4.1 Sample Configuration

A single configuration file is sufficient for configuring quite everything: the **spawnd** connection broker, **tac_plus-ng** and the *MAVIS* authentication and authorization back-end.

The daemon supports *shebang* syntax. If the configuration file is executable and starts with

```
#!/usr/local/sbin/tac_plus-ng
```

then it can be started directly.

The first step is to configure the **spawnd** portion to tell the daemon the addresses and TCP ports to listen on and to, eventually pass *realms*:

```
id = spawnd {
    listen { port = 49 }
    listen { port = 4949 }
    listen { address = ::0 port = 4950 realm = customer1 }
    listen { address = 10.0.0.1 port = 4951 realm = customer2 }
    # listen { address = 10.0.0.1 port = 4951 realm = customer2 tls = yes }
    #
    # See the spawnd configuration guide for further configuration options.
}
```

The thing that needs some explanation here is *realms*. A *realm* in **tac_plus-ng** summarizes a set of configuration options. Realms inherit configurations from their parent realm, including the parent ruleset, which will be evaluated if the local ruleset doesn't exist or doesn't return a verdict.

The default realm is internaly named `default`. Using *realms* is optional.

Now to the actual **tac_plus-ng** configuration which starts with

```
id = tac_plus-ng {
    # This is the top-level realm, actually.
```

The second line above starts a comment. 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.

Typically, the next step is to define log destinations and tell the daemon to use them. This sample logs to disk, but other destinations (syslog, pipe) are available, too.

```
    log authzlog { destination = /var/log/tac_plus/authz/%Y/%m/%d.log }
    log authclog { destination = /var/log/tac_plus/authc/%Y/%m/%d.log }
    log acctlog  { destination = /var/log/tac_plus/acct/%Y/%m/%d.log }
    accounting log = acctlog
    authentication log = authclog
    authorization log = authzlog
```

Logs are interited to sub-realms and while sub-realms can define their own logging that won't override the parent realm definitions.

You can specify a retire limit to have the server auto-terminate and restart its worker processes:

```
    retire limit = 1000
```

Then, there's the *MAVIS* part:

```
mavis module groups {
    resolve gids = yes
    resolve gids attribute = TACMEMBER
    groups filter = /^(guest|staff|ubuntu)$/
}

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

user backend = mavis
login backend = mavis chpass
pap backend = mavis
```

which defines interaction with external external back-ends.

You can define network objects for later use in ACLs:

```
net outThere { address = 100.65.3.1 address = 100.66.0.0/16 }
```

Networks can be hierarchic, too:

```
net all {
    net north {
        address = 100.67.0.0/16
    }
    net south {
        address = 100.68.0.0/16 }
}
```

Now, define device objects for your network access devices. Just like realms and networks these can be hierarchic:

```
device world {
    welcome banner = "\nHitherto shalt thou come, but no further. (Job 38.11)\n\n"
    key = QaWsEdRfTgY
    enable 15 = clear test
    address = ::/0
    device south {
        address = 100.99.0.0/16
    }
    device west {
        address = 100.100.0.0/16
    }
}

device localhost {
    address = 127.0.0.1
    welcome banner = "Welcome home\n"
    parent = world # for key and other definitions not set here
}

device rfc {
    address = 172.16.0.0/12
    welcome banner = "Welcome private\n"
    key = labKey
}
```

Now, define some profiles. These will be assigned to users later:

```
profile readwrite {
    script {
```

```
            if (service == shell) {
                if (cmd == "")
                    set priv-lvl = 15
                permit
            }
        }
    }

    profile getconfig {
        script {
            if (service == shell) {
                if (cmd == "") {
                    set autocmd = "sho run"
                    set priv-lvl = 15
                    permit
                }
            }
        }
    }

    profile engineering {
        script {
            if (service == shell) {
                if (cmd == "") {
                    set priv-lvl = 7
                    permit
                }
                if (cmd =~ /^ping/) deny
                permit
            }
        }
    }

    profile guest {
        script {
            if (service == shell) {
                if (cmd == "") {
                    set priv-lvl = 1
                    permit
                }
            }
            permit
        }
    }
```

Your can define groups to implement a role-based access control scheme ...

```
    group admin {
        group north # "admin" is a member
        group south # of both
    }

    group engineering

    group guest
```

... and add users:

```
    user demo {
        password login = clear demo
        member = engineering,admin
    }
```

```
user readonly {
    password login = clear readonly
    member = guest
}
```

You can optionally assign a profile to a user by either referencing or embedding it. This skips ruleset evaluation.

```
user demo1 {
    password login = clear demo
    profile = engineering
}

user demo2 {
    password login = clear demo
    profile {
        # Inline script definition. Alternatively, defining an ACL with the scripts
        # content and refering to that using "script = aclName" works too.
        script {
            if (service == shell) {
                if (cmd == "") {
                    set priv-lvl = 1
                    permit
                }
            }
            permit
        }
    }
}
```

Finally, implement a rule-set to assign profiles to users. As mentioned a couple of lines above, the `ruleset` will not be evaluated for users with a profile pre-assigned, so if all your users have a profile pre-assigned you may as well skip that part:

```
ruleset {
    rule from-localhost {
        enabled = yes
        script {
            if (nas == localhost) {
                if (group ==  admin) {
                    profile = admin
                    permit
                }
                if (group ==  engineering ) {
                    profile = engineering
                    permit
                }
            }
      }
    }
    rule from-rfc {
        enabled = yes
        script {
            if (nas == rfc) {
                if (group == south) {
                    profile = admin
                    permit
                }
                if (group ==  engineering ) {
                    profile = engineering
                    permit
                }
            }
```

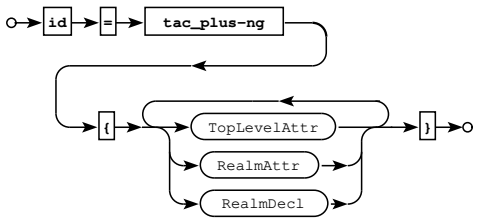
```
                }
            }
        }
}
```

4.2 Configuration directives

Configuration options include

1. global options
2. realms
3. devices
4. time specifications
5. profiles
6. groups
7. users
8. access lists
9. rules

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

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).

4.2.1 Global options

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

4.2.1.1 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

- `last-recently-used limit =` *n*

  A LRU limit can be set to prioritize new connections. If adding a connection exceeds a total of *n* connections then the least recently used connection will be closed. Set this somewhat lower than the `users max` parameter in the *spawnd* section to have any impact.

  Default: unset

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

4.2.1.2 DNS

tac_plus-ng can make use of both static and dynamic (via c-ares) DNS entries. Configuration options at global (and realm) level are:

- `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

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

- `dns cache period =` *seconds*

  This option specifies the minimum DNS response caching time (default: 1800 seconds).

- `dns servers = "` *string* `"`

  This option specifies the servers to use. This string will be evaluated by *ares_set_servers_ports_csv(3)*, please see the corresponding man page for details. The option isn't available if compiled without DNS support.

The following configuration options are available at global, realm, device and net level.

- `dns reverse-lookup[nac|nas] = (yes|no)`

  This will perform a DNS reverse lookup on the NAC address, the NAS address or (if unspecified) both.

- `dns timeout =` *seconds*

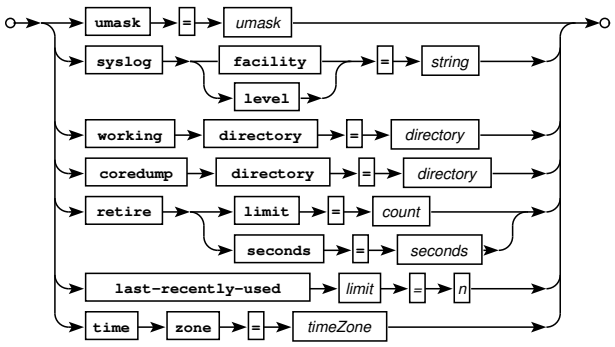
  This option specifies the maximum amount of time to wait for a DNS response.

4.2.1.3 Process-specific options

There are a couple of process-specific options available:

- `coredump directory =` *directory*
  Dump cores to *directory*. You really shouldn't need this.

4.2.1.4 Railroad Diagrams



Railroad diagram: GlobalDecl

4.3 Realms

Bascially, realms are containers to logically separate configuration sets. At top-level, there's the default realm (called `default` internally). Realms pass on most configurations (e.g. logging, users (if there are no users defined in that realm scope), groups, profiles) to their sub-realms.

Realm selection is intially based on **spawnd** configuration:

```
spawnd = {
    listen { port = 49 }   # implied realm is "default"
    listen { port = 3939 } # implied realm is "default"
    listen { port = 4949 realm = realmOne }
    listen { port = 5959 realm = realmTwo }
}
```

If *VRFs* are used and no `realm` is specified in the **spawnd** section, the daemon will try to use the *VRF name* as `realm` and fall back to the `default` realm if that "vrf realm" isn't defined.

A realm can be selected based on device address, too:

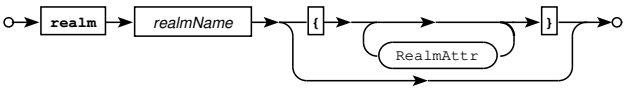
```
device myDevices { address = 10.1.23.0/24 target-realm = realmOne }
```

The syntax to use (and define) realms is

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

at top configuration level. Realms cover devices, users, groups, profiles, rulesets, times, *MAVIS* configurations other configuration options.

4.3.1 Railroad Diagrams



Railroad diagram: RealmDecl

4.4 Realm attributes

The following options may be specified at *realm* level. This includes the `default` *realm*:

4.5 Logging

Logging options defined in the top-level `default` realm will be shared with sub-realms unless the sub-realm has its own logging configuration. The software provides logs for

- Authentication

```
authentication log = log_destination
```

- Authorization

```
authorization log = log_destination
```

- Accounting

```
accounting log = log_destination
```

- Connections

```
connection log = log_destination
```

- RADIUS Authentication

```
radius.acccess log = log_destination
```

- RADIUS Accounting

```
radius.accounting log = log_destination
```

Logs may be written to multiple destinations.

Valid log destinations are *named*, e.g.:

```
log mylog {
  destination = 169.254.0.23       # UDP syslog, default port
  # destination = 169.254.0.24:514 # UDP syslog, port specified
  # destination = [fe80::123:4567:89ab:cdef]     # IPv6 UDP
  # destination = [fe80::123:4567:89ab:cdef]:514 # IPv6 UDP, port specified
  # destination = "/tmp/x.log"     # plain file, async writes
  # destination = ">/tmp/x.log" # plain file, sync writes
  # destination = "|my_script.sh"  # script
  # destination = syslog           # syslog(3)
  #
  syslog facility = MAIL           # sets log facility
  syslog level = DEBUG             # sets log level
  # syslog ident = tacplus
  #
  # # IP source spoofing for UDP syslog. This will require root privileges, or
  # # (on Linux) CAP_NET_RAW capabilities.
  # # Requires a dedicated destination per protocol:
  # destination = 169.254.0.24:514,[fe80::123:4567:89ab:cdef]:514
  # source spoofing = yes
  # # This is basically similar to tacspooflog-ng.pl funcctionality. Alas, this
  # # comes with some restrictions also, e.g. on Linux it may not work via the
  # # lo interface. Also, only a subset of the other platforms has been tested.
```

```
  #
  # timestamp = RFC3164 # default for syslog, other options are
  #                     # "RFC5424" or "none", see the TIMESTAMP
  #                     # log variable for details.
  #
  # buffer size = 64000 # default upper limit for log buffer, use 0 to unlimit
  #
  # Logs may be filtered using a script:
  # filter = logfilter # pre-existing script
  # filter { if (user =~ /^demo/) deny } # inline script
  #
  #
  # Also, you can rewrite logs. E.g.: Define this at realm level:
  # rewrite sanitize {
  #   rewrite /(key|secret|password) (\d) (\w+)/ "$1 $2 ***"
  #   rewrite /(key|secret|password) (\w+)/ "$1 ***"
  # }
  #
  # Then use
  #   rewrite = sanitize
  # in log context.
}
authentication log = mylog
accounting log = mylog
authorization log = mylog
```

Syslog
Logging non-session related output to `syslogd`*(8)* can be disabled using

```
syslog default = deny
```

Also, a log filter can be set using `syslog filter` = *number*, where *number* is a binary `OR` of message class values:

Bit	Value	Class (and sample message)
0	1	Authentication (`... shell login for ...`)
1	2	Certificates (`... peer certificate ... will expire...`)
2	4	Connections (`... connection request from ...`)
3	8	Keys/secrets (`... uses deprecated key`)
4	16	MAVIS (`... looking for ... in MAVIS backend... `)
5	32	Single-connection (`... possibly no single-connection ...`)

The default is a binary `OR` of all of those values, so no log messages are being suppressed.
So, if you wouldn't want authentication or mavis logs to show up in syslog you could just sum up and set the remaining values:

```
syslog filter = 46 # 2 + 4 + 8 + 32
```

Please note that filtering actual error messages is not supported.

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

```
    destination = /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).

A couple of other configuration options that may be useful in `log` context include:

- `(authentication|authorization|accounting radius.authorization|radius.accounting)format` = *string*

This defines the logging format. `strftime(3)` conversions are recognized. The following variables are resolved:

`${cmd}`, `${cmd,`*separator*`}`	values of `cmd=` and `cmd-arg=` attribute-value pairs, separated by whitespace or *separator*. Will be mapped to `${args}` if `service` isn't `shell`
`${args}`, `${args,`*separator*`}`	input attribute-value pairs (excluding `service`, separated by whitespace or *separator*
`${rargs}`, `${rargs,`*separator*`}`	output attribute value pairs, separated by whitespace or *separator*
`${device.address}`, `${nas}` [deprecated]	Device IP address
`${device.dnsname}`, `${nas-name}` [deprecated]	Device DNS reverse mapping
`${device.name}`, `${host}` [deprecated]	Device name of matching device declaration
`${device.port}`	NAS port (console, tty, ...)
`${client}`, `${nac}`	Client IP address
`${user}`	user name
`${user.original}`	user name before any rewrite operations
`${profile}`	profile assigned to user
`${service}`	service type (e.g. `shell`)
`${result}`	typically `permit` or `deny`
`${hint}`	`added`/`replaced` for authorization, informal text for accounting, error message for TLS failures
`${client.dnsname}`, `${nac-name}` [deprecated]	Client DNS reverse mapping
`${msgid}`	A message ID, perhaps suitable for RFC5424 logs. These are listed somewhere below.
`${type}`	packet type (`authen`/`author`/`acct`)
`${accttype}`	accounting type (`start`/`stop`/`update`)
`${priority}`	syslog priority
`${action}`	authentication info (e.g. `pap login`)
`${privlvl}`	privilege level
`${authen-action}`	`login` or `chpass`
`${authen-type}`	Authentication packet type, e.g. `AUTHEN/PASS`, `AUTHEN/FAIL`
`${authen-service}`	`asciiascii`/`pap`/`chap`/`mschap`/`mschapv2`
`${authen-method}`	`krb5`/`line`/`enable`/`local`/`tacacs+`/`guest`/`radius`/`krb4`/`rcmd`
`${rule}`	Name of the matching rule.
`${label}`	Ruleset label, if any.
`${config-file}`	Configuration file name
`${config-line}`	Configuration file line number
`${context}`	Context variable (set via `context = ...`)
`${vrf}`	Name of the current socket IPv4 vrf, supported on Linux (requires `sysctl net.ipv4.tcp_l3mdev_accept=1`) and possibly OpenBSD.
`${realm}`	realm name
`${uid}`	UID from PAM backend
`${gid}`	GID from PAM backend
`${gids}`	GIDs from PAM backend
`${home}`	Home directory from PAM backend
`${shell}`	Shell from PAM backend
`${dn}`	Raw `dn` backend value, typically from LDAP
`${ident}`	The `ident` string from log context (default: `tacplus`)
`${identity-source}`	The `IDENTITY_SOURCE` backend value (the *identitySourceName* of the originating *MAVIS* module)
`${log-sequence}`	Log sequence number
`${mavis.latency}`	The milliseconds it took the *MAVIS* backend to answer a request
`${memberof}`	Raw `memberOf` backend value, typically from LDAP
`${pid}`	Process ID

`${peer.address}`	Communication peer IP address
`${peer.port}`	Communication peer TCP/UDP port
`${server.name}`, `${hostname}` [deprecated]	Server host name
`${server.address}`	Server address
`${server.port}`	Server TCP or UDP port
`${session.id}`	Session id
`${tls.conn.version}`	TLS Connection Version (requires OpenSSL)
`${tls.conn.cipher}`	TLS Connection Cipher (requires OpenSSL)
`${tls.peer.cert.issuer}`	TLS Peer Certificate Issuer (requires OpenSSL)
`${tls.peer.cert.sha1}`	TLS Peer Certificate SHA1 fingerprint (requires OpenSSL)
`${tls.peer.cert.sha256}`	TLS Peer Certificate SHA256 fingerprint (requires OpenSSL)
`${tls.peer.cert.subject}`	TLS Peer Certificate Subject (requires OpenSSL)
`${tls.conn.cipher.strength}`	TLS Connection Cipher Strength (requires OpenSSL)
`${tls.peer.cn}`	TLS peer certificate Common Name (requires OpenSSL)
`${tls.psk.identity}`	TLS PSK identity (requires OpenSSL)
`${tls.conn.error}`	OpenSSL reason for a failed TLS handshake, e.g. tlsv1 alert unknown ca (requires OpenSSL)
`${tls.conn.error.code}`	OpenSSL error code for a failed TLS handshake, in hex, e.g. 0A000418 (requires OpenSSL)
`${tls.peer.cert.verify}`	Peer certificate verification error for a failed TLS handshake, e.g. certificate has expired (requires OpenSSL)
`${tls.peer.cert.verify.code}`	Peer certificate verification error code (X509_V_ERR_*) for a failed TLS handshake, e.g. 10 (requires OpenSSL)
`${FS}`	Field separator, default is `|` for syslog, `\t` else
`${TIMESTAMP}`	`strftime`*(3)* timestamp format, `"%b %e %H:%M:%S"` for `timestamp = RFC3164` (default for syslog) `"%Y-%m-%dT%H:%M:%S.%06N%:z"` for `timestamp = RFC5424`, `"%Y-%m-%d %H:%M:%S %z"` else.

The pre-set defaults for logging are:

```
accounting format = "${nas}${FS}${user}${FS}${port}${FS}${nac}${FS}${acctype}${FS}${ ↩
    service}${FS}${cmd}"
authorization format = "${nas}${FS}${user}${FS}${port}${FS}${nac}${FS}${profile}${FS}${ ↩
    result}${FS}${service}${FS}${cmd}"
access format = "${nas}${FS}${user}${FS}${port}${FS}${nac}${FS}${action} ${hint}"
connection format = "${acctype}${FS}${conn.protocol}${FS}${peer.address}${FS}${peer.port} ↩
    ${FS}${server.address}${FS}${server.port}${FS}${tls.conn.version}${FS}${tls.peer.cert. ↩
    issuer}${FS}${tls.peer.cert.subject}"
radius.access format = "${nas}${FS}${user}${FS}${port}${FS}${nac}${FS}${acctype}${FS}${ ↩
    action} ${hint}${FS}${args, }${FS}${rargs, }"
radius.accounting format = "${nas}${FS}${user}${FS}${port}${FS}${nac}${FS}${acctype}${FS} ↩
    ${service}${FS}${args, }"
```

Also, there are certain default prefix and postfix settings for strings to prepend or append to the format:

Log destination is UDP syslog:

```
separator = "|" # that's the ${FS} variable above
prefix = "<${priority}>${TIMESTAMP} ${hostname} ${ident}[${pid}]: ${msgid}${FS}"
postfix = ""
```

Log destination is syslog(3):

```
separator = "|"
prefix = "${msgid}${FS}"
postfix = ""
```

Log destination is a file or a pipe:

```
prefix = "${TIMESTAMP} "
postfix = "\n"
separator = "\t"
```

Message ID	Description
ACCT-START	accounting start
ACCT-STOP	accounting stop
ACCT-UNKNOWN	unknown (non-compliant) accounting data
ACCT-UPDATE	accounting update/watchdog
AUTHC-FAIL	generic authentication failure
AUTHC-FAIL-ABORT	authentication was aborted
AUTHC-FAIL-BACKEND	the authentication backend failed
AUTHC-FAIL-BUG	authentication failed due some programming error
AUTHC-FAIL-DENY	authentication was denied
AUTHC-FAIL-WEAKPASSWORD	the password used didn't met minimum criteria
AUTHC-FAIL-ACL	access was denied due to ruleset or acl
AUTHC-FAIL-DENY-RETRY	the user tried the same wrong password once more
AUTHC-FAIL-PASSWORD-NOT_TEXT	the password isn't specified as clear-text
AUTHC-FAIL-BAD-CHALLENGE-LENGTH	the MSCHAP challenge length didn't match
AUTHC-FAIL-NOPASS	there's no passwort set for the user
AUTHC-FAIL-BADSECRET	the wrong RADIUS secret was used
AUTHC-PASS	authentication passed
AUTHZ-PASS	authorization succeeded
AUTHZ-PASS-ADD	authorization succeeded, attribute-value-pairs were added
AUTHZ-PASS-REPL	authorization succeeded, attribute-value-pairs were replaced
AUTHZ-FAIL	authorization failed
CONN-REJECT	connection was rejected
CONN-START	connection was started
CONN-STOP	connection was terminated

- `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
```

4.5.1 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 default for text files is:

- 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.

Attributes, as sent by the NAS, might be:

```
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 appear,. randomly..

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.

4.5.2 Spoofing Syslog Packets

The script `tacspooflog-ng.pl` (which comes bundled with this distribution, have a look at the `tac_plus-ng/extra/` directory) may be used to make `syslogd` believe that logs come straight from your router, not from `tac_plus-ng`.

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

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

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

In contrast to the older `tacspooflog.pl` script `tacspooflog-ng.pl` will handle both IPv4 and IPv6 addresses and has more configuration options.

Please read `tac_plus-ng/extra/tacspooflog-ng.README` too, if you're thinking about using this script.

4.6 User Messages

User messages, e.g. the `Username` prompt, can be customized, both at device and realm level:

```
message USERNAME = "Utilisateur: "
```

Supported messages and their defaults:

ID	Default value
ACCOUNT_EXPIRES	"This account will expire soon."
BACKEND_FAILED	"Authentication backend failure."
CHANGE_PASSWORD	"Please change your password."
DENIED_BY_ACL	"Denied by ACL"
ENABLE_PASSWORD	"Enable Password: "
PASSWORD	"Password: "
PASSWORD_ABORT	"Password change dialog aborted."
PASSWORD_AGAIN	"Retype new password: "
PASSWORD_CHANGE_DIALOG	"Entering password change dialog"

ID	Default value
PASSWORD_CHANGED	"Password change succeeded."
PASSWORD_EXPIRED	"Password has expired."
PASSWORD_EXPIRES	"Password will expire on %c." (fed to *strftime*(3))
PASSWORD_INCORRECT	"Password incorrect."
PASSWORD_MINREQ	"Password doesn't meet minimum requirements."
PASSWORD_NEW	"New password: "
PASSWORD_NOMATCH	"Passwords do not match."
PASSWORD_OLD	"Old password: "
PERMISSION_DENIED	"Permission denied."
RESPONSE	"Response: "
RESPONSE_INCORRECT	"Response incorrect."
USERNAME	"Username: "
USER_ACCESS_VERIFICATION	"User Access Verification"

The PASSWORD prompt of an ASCII login may also be set by a rule, see the message PASSWORD action in "Syntax" (scripts).

4.7 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

- `fragment timeout =` *s*

  Terminate a connection to a NAS if fragmented packets aren't completed in time.

  Default: 3

- `context timeout =` *s*

  Clears context cache entries after *s* seconds of inactivity. Default: 3600 seconds.

  Default: 3600

  This configuration will be accepted at realm level, too.

- `warning period =` *d*

  Set warning period for password expiry to *d* days.

  Default: 14

- `max-rounds =` *n*

  This sets an upper limit on the number of packet exchanges per session. Default: 40, acceptable range is from 1 to 127.

4.7.1 Authentication

- `password acl =` *acl*

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

Authentications using passwords that fail the check will be rejected.

- `password max-attempts` = *integer*

  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.

Thid directive can be used at device level, too.

- `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 being unset.

  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 simply resend the user password at the `Enable Password:` prompt. In that case you've got only two options: Either try

```
    enable = login
```

  at user profile level, which will omit the secondary password query and let the user enable with his login password, or permit anonymous `enable` with

```
    anonymous-enable = permit
```

  in device context to use the enable passwords defined there.

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

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

There are also experimental options for (non-standard) SSH public key authentication available. These may or may not supported by your vender:

- `ssh-key =` *public-ssh-key-in-OpenSSL-authorized_keys-format*

  Example: `ssh-key = "AAAAB3NzO4S6C/SAu9E90P3n9dfbe3iNiK...STPC6V1fffa123OxmK3hhzwbl"`

- `ssh-key-hash =` *ssh-key-in-OpenSSL-format*

  There's no use in specifying the hash if you've configured the public key, the daemon will care for that itself.

  Example: `ssh-key-hash = SHA256:kOkclqivcjludf/jdsfkyqpddffdk38U12+CkA8fBAC`

4.7.2 User back-end options

These options are relevant for configuring the MAVIS user back-end:

- `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 device level.

- `user backend = mavis`

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

- `pap backend = mavis [ prefetch ]`

  Verify PAP passwords using the MAVIS back-end. This needs to be set to either `mavis` or `mavis prefetch` in order to authenticate PAP requests using the MAVIS back-end. 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 back-end 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 back-end. This needs to be set to either `mavis` or `mavis prefetch` in order to authenticate login requests using the MAVIS back-end. 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 back-end 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 back-end modules.

- `chap backend = mavis [ prefetch ]`

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

  `prefetch` is actually enabled automatically. The daemon will first retrieve the users' profile from the back-end and then authenticate the user based on information eventually found there.

  This directive implies `user backend = mavis`.

- `ms-chap backend = mavis [ prefetch ]`

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

  `prefetch` is actually enabled automatically. The daemon will first retrieve the users' profile from the back-end and then authenticate the user based on information eventually found there.

  This directive implies `user backend = mavis`.

- `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` = *acl*

  Query MAVIS user back-end only if *acl* matches. Defaults to:

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

- `mavis format` *attribute* = *string*

  Valid attributes are `custom_0` to `custom_3`. The corresponding MAVIS attributes will be set for MAVIS requests. Log variable substitution applies. For example,

```
mavis format custom_0 = "${server.address} ${server.port}"
```

  will set the AV_A_CUSTOM_0 MAVIS attribute.

  Tip: custom attributes can also be in ACLs. Example: `custom_0 = "string"`.

- `mavis mfa` = (`yes` | `no`)

  `mavis mfa acl` = *acl*

  Authenticate users via MAVIS MFA. These directives work in user context, too. Also, you may use them in scripts (for authorizing shell access, for example.)

4.7.3 TLS and DTLS

TACACS+ over TLS1.3 is supported, as is RADIUS over both TLS and DTLS. Availability of toese features may depend on your OpenSSL version.

The following configuration options are availablet:

- `tls cert-file` = *cert-file*

  Specifies the public part of a TLS server certificate in PEM format.

- `tls key-file` = *key-file*

  Specifies the private part (the key) of a TLS server certificate in PEM format.

- `tls passphrase` = *passphrase*

  Specifies the optional passphrase to decrypt *key-file*.

- `tls accept expired` = (`yes` | `no`)

  Accept expired certificates.

- `tls verify-depth` = *depth*

  Sets TLS verification depth.

- `tls cafile` = *cafile*

  Specifies a file with the CAs to use.

- `tls alpn` = *ALPN-Protocol-ID*

  There's currently no ALPN Protocol ID registered for TACACS-over-TLS, the official list is here: TLS Application-Layer Protocol Negotiation (ALPN) Protocol IDs.

- `tls auto-detect` = (`yes`|`no`)

  Enable TLS auto-detection. Defaults to `no`.

- `tls.peer.cert.sha1` = *SHA1-certificate-fingerprint*

  `tls.peer.cert.sha256` = SHA256 certificate fingerprint

  Certificate fingerprints can be used for certificate-based device identification without a certification authority. Fingerprints need to be unique. Example:

```
host demo {
    tls.peer.cert.sha256 = 00:01:02:03:04...
    tls.peer.cert.1 = 0001020304... # the colons are optional
}
```

- `tls.peer.cert.validation` = (`none`|`any`|`hash`|`cert`)

  You can choose whether to skip certificate validation (`none`), to check the fingerprint only (`hash`), to validate the full chain (`cert`) or to allow both of the latter (`any`, that's the default).

- `tls.peer.cert.san.validation` = (`yes`|`no`

  This option enables SAN IP validation. Defaults to `no`, but enabling it will skip device object selection based on the SAN IP if the latter doesn't equal the peer IP. Not sure whether there's a use case for this at all.

- `tls cipher-suites` = *string*

  `tls key-exchange` = *string*

  `tls dhe` = *string*

  `tls groups` = *string*

  `tls supported-groups` = *string*

  Sets the corresponding OpenSSL parameters.

If compiled with OpenSSL support, TLSv1.2 and TLSv1.3 Preshared Keys and SNIs are supported:

- `tls psk` = (`yes`|`no`)

  This enables PSK support at realm level, if supported by your OpenSSL version. Please note that PSK and cert-based authentication are mutually exclusive in OpenSSL, and a dedicated listener configuration with `tls psk = yes` in **spawnd** context might be required. Exception: For TLS1.3, **tac_plus-ng**will permit for PSK and certificate authentication on the same port by evaluating the initial ClientHello message.

  PSK identity and key can be declared at device or realm level:

```
tls psk id = myid
#tls psk key = clear demoDemo # plain text
#tls psk key = 6 'FFM^eVF\gQS^ZeW`TYeaHeD^eKPMgXAAB' # type 6, requires master key
#tls psk key = 7 2703130E0E725C555C # type 7
#tls psk key = base64 'ZGVtb0RlbW9EZW1vCg==' # base64
tls psk key = 0123456789abcdef # hex
```

- `tls sni` = *SNI*

  This adds *SNI* to the server name list of the current realm. A TLS connection requesting *SNI* will automatically be mapped to that realm.

Example:

```
id = spawnd {
    listen { port = 49 realm = heck }
    listen { port = 300 realm = heck tls = yes }
    listen { port = 301 realm = heck tls = yes tls psk = yes}
    spawn { instances min = 1 instances max = 32 }
    id = tac_plus-ng {
            ...
        realm heck {
            tls cert-file = /somewhere/tac-ca/server.tacacstest.crt
            tls key-file = /somewhere/tac-ca/server.key
            tls ca-file = /somewhere/tac-ca/ca.crt
            ...
        }
    }
}
```

4.8 Miscellaneous

In realm context:

- `haproxy auto-detect =( yes | no )`
  Enable HAProxy protocol v2 auto-detection. Defaults to `no`.

In **spawnd listen** context,

- `haproxy =( yes | no )`
  will will tell **tac_plus-ng** to auto-detect that a connection is proxied via HAProxy protocol 2.
  A suitable HAProxy configuration could look similar to:

```
frontend tacplus
    bind *:49
    mode tcp
    default_backend backendtacplus

backend backendtacplus
    balance source
    server tacserver1 127.0.0.1:4949 no-check send-proxy-v2
```

- `tls =( yes | no )`
  will tell **tac_plus-ng** whether the connection is TLS encrypted.
- `vrf =` (*vrf-name* | *vrf-number*)
  will tell **spawnd listen** to *bind(2)* the requested VRF (*vrf-name* on Linux, *vrf-number* on OpenBSD).

Example:

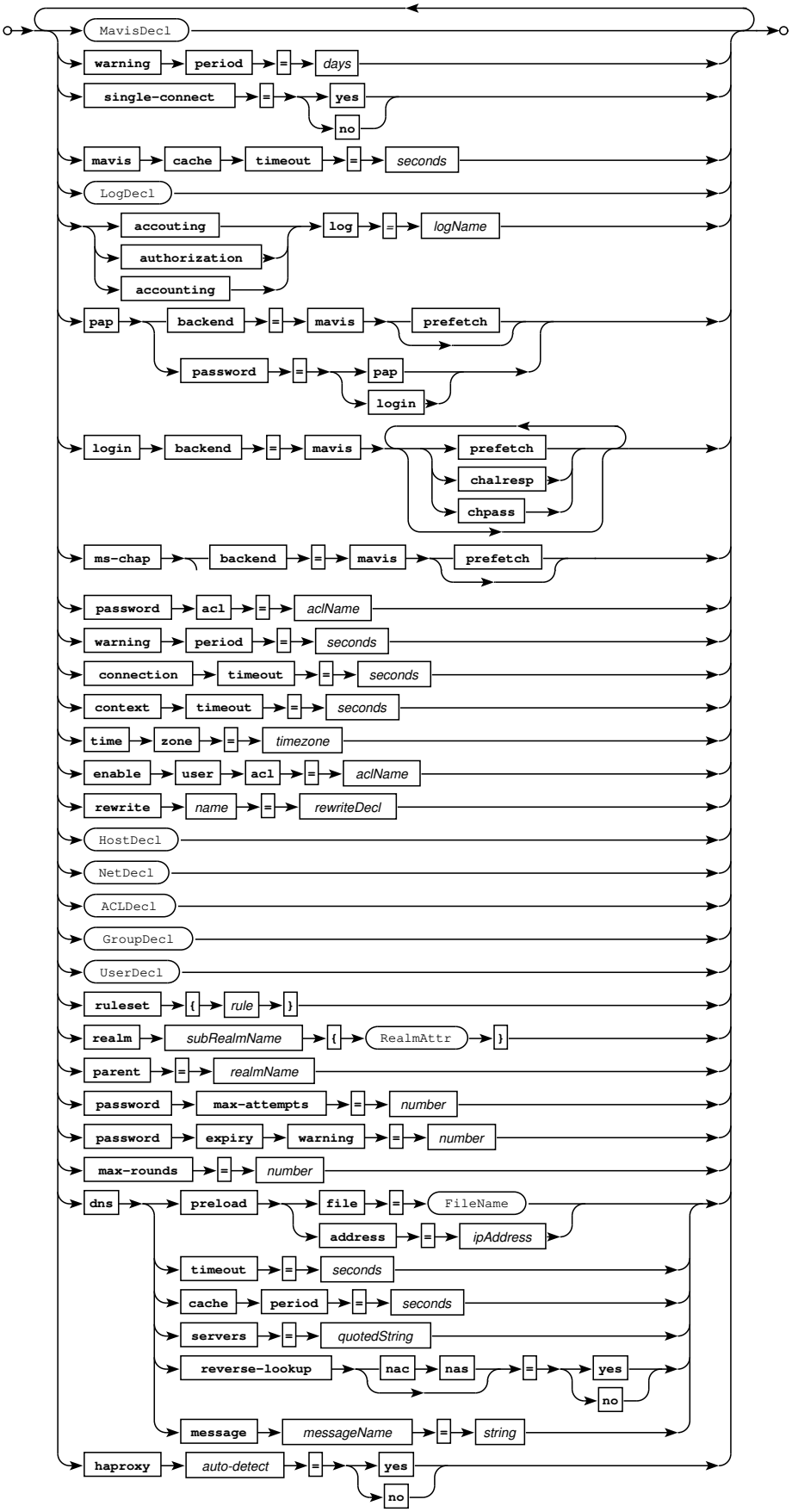
```
id = spawnd {
    ...
    listen {
        port = 49
        vrf = vrf-blue
        tls = true
        haproxy = true
    }
    ....
}
```

4.9 Realm Inheritance

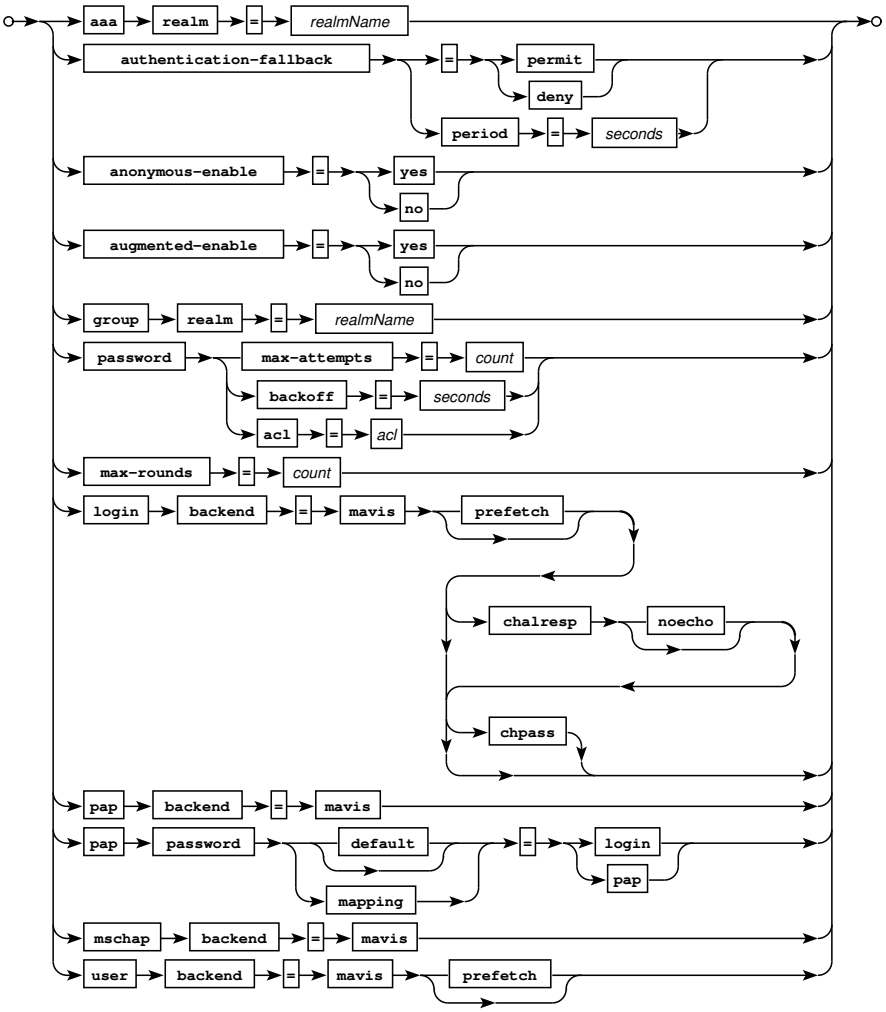
Realms inherit quite some configuration from their parent realm:

Declaration of ...	**is taken from parent realm ...**
acl	if not found in current realm
dns forward mapping	if not found in current realm
group	if not found in current realm
device (IP lookup)	if no device defined in current realm
device (name lookup)	if not found in current realm
log	always
mavis module	if not set and no users defined in current realm
network	if not found in current realm
profile	if not found in current realm
ruleset	if not set or undefined result in current realm
time	if not found in current realm
user	if not found in current realm

4.10 Railroad Diagrams



Railroad diagram: RealmAttr



Railroad diagram: RealmAttrAuthen

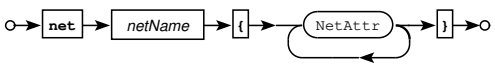
4.11 Networks

Networks consist of IP addresses or other networks. They may overlap. Networks can be used in ACLs. The parent of a network may be set either implicitly (by defining it it parent context) or explicitly.

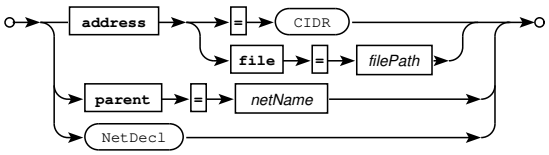
```
net home {
      address = 172.16.0.0/23
      net dev {
          address = 172.16.0.15
      }
      parent = ...

}
```

4.11.1 Railroad Diagrams



Railroad diagram: NetDecl



Railroad diagram: NetAttr

4.12 Devices (Hosts)

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

If you want **tac_plus-ng** 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

```
device world4 {
  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 devices 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 device 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 device object attributes (key, prompt, enable passwords) may only exist once.

Aggregation
Some IP lookups support aggregation. E.g., defining

```
    aggregate device = 172.16.0.0/23
```

at realm level will map `device` related IP lookups in that range to `172.16.0.0`.
Likewise, `net` related lookups use aggregates, too:

```
    aggregate net = 192.168.0.240/255.255.0.240
```

Aggregation is supported for both IPv4 and IPv6. Also, netmasks don't need to be contiguous.

If compiled with TLS support, primary criteria for device object selection with TLS is no longer the NAS IP address but the certificate DNS SAN (order: IP, directory name, common name), followed by the DN and, common name. Sample DN matching sequence for `/C=US/ST=California/L=San Francisco/O=Example Corp/OU=IT Department/CN=www.example.`
The daemon will check for device objects named

- `/C=US/ST=California/L=San Francisco/O=Example Corp/OU=IT Department/CN=www.example.com`,
- `/C=US/ST=California/L=San Francisco/O=Example Corp/OU=IT Department`,
- `/C=US/ST=California/L=San Francisco/O=Example Corp`,
- `/C=US/ST=California/L=San Francisco`,
- `/C=US/ST=California`,
- `/C=US`, followe by
- `www.example.com`,
- `example.com` and
- `com`

before falling back to IP based selection.

If your DN fields are in reverse order field stripping starts at the beginning of the DN, not at the end.

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

```
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 device declarations conforms to

`device`*name* { *key-value pairs* }

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

- `key` [`warn`] (*YYYY−MM−DD* | *s*)] = [(`clear` | `6` | `7`)] *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 = ""
  ```

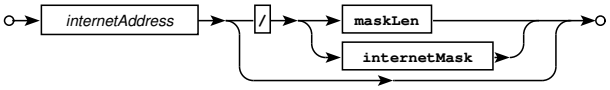
  Be careful to remember to switch encryption back on again after you've finished debugging.

- `radius.key` [`warn`] (*YYYY−MM−DD* | *s*)] = [(`clear` | `6` | `7`)] *string*

  The RADIUS secret can be set using this directive.

- `address` = *cidr*

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



Railroad diagram: CIDR

- `address file` = *file*

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

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

  This directive may be used to permit or deny the single-connection feature for a particular device 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.

  This configuration will be accepted at realm level, too.

- `parent =` deviceName

  This sets the the parent devices. Definitions not found in the current device will be looked up there, recursively.

- `device` deviceName `{` DeviceAttr `}`

  Devices can be defined in device context, too.

- `script` (= *aclName* | `{` *tacAction* `}`)

  Scripts can be used in device context. These are run before AAA and may be used to permit or deny access, or to rewrite usernames.

  This configuration will be accepted at realm level (for the *default host*, too.

4.12.1 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.

4.12.2 Authentication

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

- `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.

- `enable [` *level* `] = ( permit | deny | error | login | ( clear | crypt | pbkdf2 | asa | 4 | 5 | 6 | 7 | 8 | 9 )` *password*)

  This directive may be used to set device 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 device level have a lower precedence as those defined at user or profile level.

  > **Password Hashes**
  > You can use the `openssl passwd` utility to compute `crypt` type password hashes.
  > The `pbkdf2` and `asa` hashes can be copied from a Cisco ASA plain text configuration. Likewise, the `4`, `5`, `7`, `8` and `9` type password hashes can be copied from IOS.

Password Encryption
If compiled with OpenSSL support Cisco Type-6 AES128 password encryption is available. Encrypted password strings can be copied from a IOS system or generated manually (for example with type6). Type-6 passwords used in host context require a *type6-key* definition for the master key in host or realm context, while *type-6* passwords in user or profile context use the master key from the realm context.

```
master key = demo
# master key = "${SOME_ENVIRONMENT_VARIABLE}"

user ... {
    password login = 6 'SeIF^JbTBWUcRIhgP[FW\'L\AAB' # no need to escape characters,  ↩
        just copy here as-is
}
```

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.

  This configuration will be accepted at realm level, too.

- `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.

  This configuration will be accepted at realm level, too.

- `password max-attempts =` *integer*

  The `max-attempts` parameter limits the number of `Password:` prompts per TACACS+ session at login. It currently defaults to `1`.

  This configuration will be accepted at realm level, too.

- `password expiry warning =` *number*

  A password expiry warning will be displayed to the user if less than *number* time is left. Appending `s` (that's the default) or any of `m`, `h`, `d`, `w` will scale the number up as expeced.

  This configuration will be accepted at realm level, too.

- `mavis backend = (yes|no)`

  Setting this to `yes` will enables MAVIS host lookups. Defaults to `no`, and it's not inherited from parent hosts.

4.12.3 Authorization

The following authorization related directives are available at device 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).

  This configuration will be accepted at realm level, too.

4.12.4 Banners and Messages

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

- `welcome banner` (*fallback*) = *string*
- `motd banner` = *string*
- `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 device, user or group level.

  These configurations will be accepted at realm level, too.
- `failed authentication banner` = *string*

  The `failed authentication banner` gets displayed upon final failure of an authentication attempt.
- `message` = *string*

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

Context	Directive	Telnet	SSHv1	SSHv2
`device`	`welcome banner`	displayed before `Username:`	not displayed	displayed before `Password:`
`device`	`reject banner`	displayed before closing connection	not displayed	not displayed
`device`	`motd banner`	displayed after successful login	not displayed	displayed after successful login
`user` or `group`	`message`	displayed after `motd banner`	not displayed	displayed after `motd banner`

A `motd banner` set by a script (see "Access Control Lists") takes precedence over the device 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 the same conversions as logs, unless specified as user level.

Example:

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

4.12.5 Workarounds for Client Bugs

The directive

`bug compatibility` = *value*

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

Bit	Value	Description
0	1	According to RFC8907 the data field should be ignored for ASCII authentications. Alas, IOS-XR puts the password exactly there. Set this if required.
1	2	Accept version 1 for authorization and accounting packets, seen with Palo Alto systems.

Bit	Value	Description
2	4	Accept key-based packet obfuscation for TLS (this violates `draft-ietf-opsawg-tacacs-tls13-24.txt`).
3	8	Accept TACACS+ payloads lower than advertized in the TACACS+ header.
4	16	Check for a space in the remote address field of a TACACS+ packet and strip everything after before attempting an IP address conversion.
5	32	Accept RADIUS packets if no Message-Authenticator attribute is set. This disables RADIUS BLAST attack migigation.
6	64	Don't require key-based packet obfuscation for non-TLS TACACS+ (this violates `draft-ietf-opsawg-tacacs-tls13-24.txt`).
7	128	Don't require TLS1.3 TACACS+-over-TLS (this violates `draft-ietf-opsawg-tacacs-tls13-24.txt`).
8	256	Ignore TACACS+ authorization AV pairs that lack the =/* separator.

Example:

```
device ... {
  ...
  bug compatibility = 2
  ...
}
```

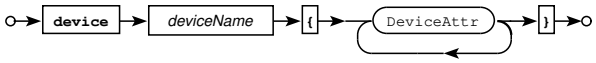
This configuration will be accepted at realm level, too.

The bug compatibility value is inherited from the corresponding parent. You can stop that with `bug compatibility = none`.

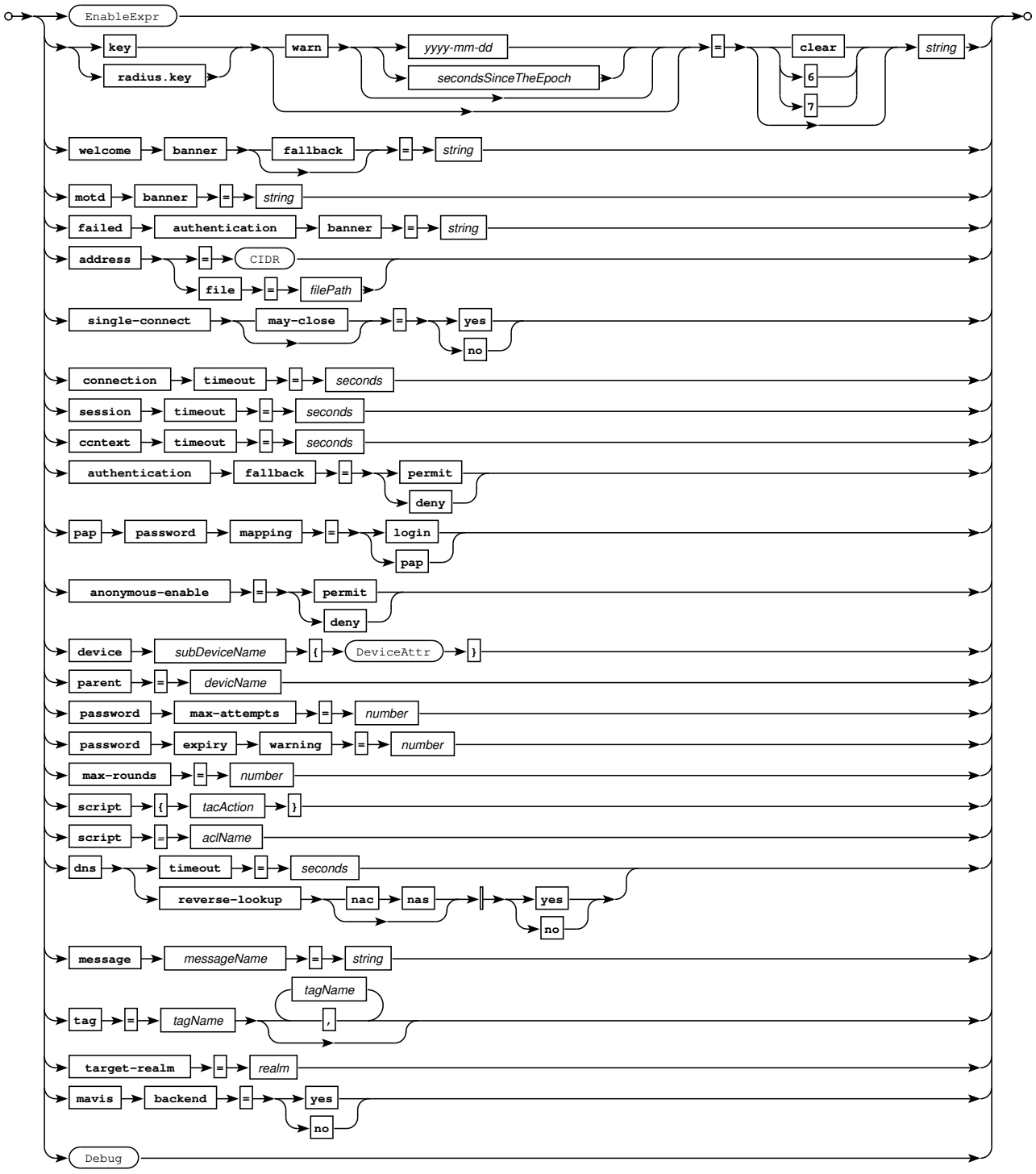
4.12.6 Inheritance and Devices

For address based device lookups, the daemon looks for the most specific device definition. Values that aren't defined (if any) will be lookup up in the device's parent, which may be either set implicitely by defining a device in the context of it's parent device, or expliitely, using the `parent` statement.

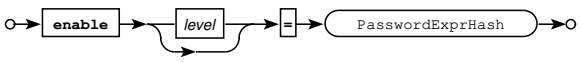
4.12.7 Railroad Diagrams



Railroad diagram: DeviceDecl



Railroad diagram: DeviceAttr



Railroad diagram: EnableExpr

4.12.8 Example

```
device customer1 {
    address = 10.0.0.0/8
```

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

device 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
    welcome banner = "\nGo away.\n\n"
}
```

4.13 Time Ranges

`time` 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:

`time` =*time_name* { "*entry*" [...] }

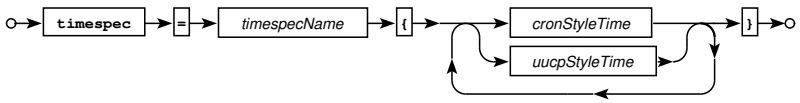
Example:

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

time sunday { "* * * * 0" }

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

4.13.1 Railroad Diagrams



Railroad diagram: TimeDecl

4.14 Access Control Lists

Access Control Lists (or, more exactly, Access Control Scripts) are the main component of ruleset evaluation.

Scripts may currently be used for ACLs, in host and profile declaration scope and in rule sets. If a script in a hierarchy doesn't return a final verdict (these are `permit` and `deny`), other scripts in the hierarchy may be evaluated. Default evaluation order is

```
script-order host = bottom-up
script-order realm = bottom-up
script-order profile = bottom-up
```

but you may prefer to change that to `top-down` to have parent scripts executed first.

To provide an example for that: In

```
profile A {
    script { ... }
    profile B {
        script { ... }
    }
```

the `script` part from `A` will by default (`bottom-up` be evaluated, if the `B script` result isn't final.

In contrast, for

```
script-order profile = top-down
profile A {
    script { ... }
    profile B {
        script { ... }
    }
```

the `A` part takes precedence and the `B script` will one be evaluated if the `A` result isn't final.

`skip parent-script = yes` may be used (at profile, host and realm level) to ignore scripts defined at a higher hierarchy level.

Scripting examples:

- `acl` *aclName* `{` tac_action ... `}`

  Example:

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

- `script` (= *aclName* | `{` *tacAction* `}`)

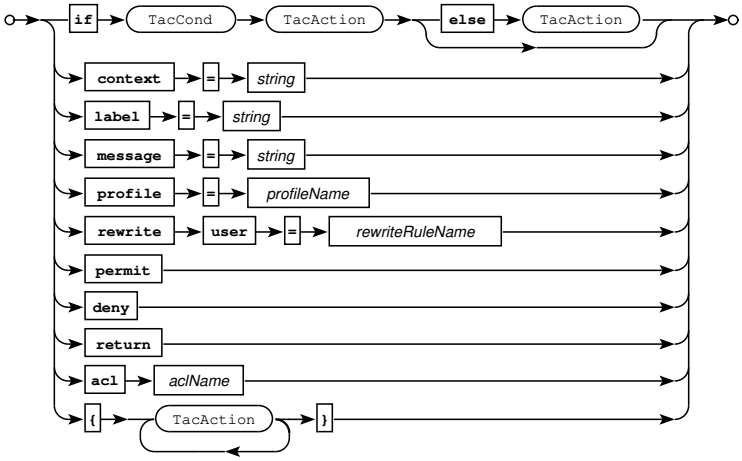
  Example:

  ```
  profile tunnelAdmin {
      script {
          if (service == shell) {
  if (cmd == "") permit # required for shell startup
              if (cmd =~ /^(no\s)?shutdown\s/) permit
              if (cmd =~ /^interface Tunnel/) permit
              deny
          }
      }
  }

  user joe {
      password = ...
      member = ops
  }
  ruleset {
      rule opsRule {
          script {
              if (group == ops)
                  profile = tunnelAdmin
                  permit
          }
      }
  }
  ```

4.14.1 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.

The message PASSWORD = string action sets the password prompt of an ASCII login, taking precedence over the PASSWORD user message of the device and realm (see "User Messages"). It's honoured in the ruleset only, by a separate pass that runs once before the prompt is sent. That pass evaluates conditions, ACLs and the permit, deny and return actions, skips everything else and discards its result; profile, rule name and profile cache are left to the regular evaluation after authentication. With MAVIS users, groups are known before the prompt only with login backend = mavis prefetch. Enable and PAP are not affected.

Example:

```
rule otp_users {
    script {
        if (member == otp) {
            message PASSWORD = "Password+OTP: "
            profile = admin
            permit
        }
    }
}
```

A rule specific prompt tells whoever enters a user name which rule that name matches, before any password is checked.

The `motd banner =` *string* action sets the motd banner for the session, see "Banners and Messages". It takes precedence over the device motd banner. The device script is evaluated at authentication time, then the ruleset, then the acl of the profile assigned by the ruleset; the last motd banner set wins. Profile scripts are evaluated for authorization and accounting only, too late for the login dialogue.

Example:

```
rule netadmins {
    script {
        if (member == netadmin) {
            motd banner = "Change freeze until Monday 06:00 UTC.\n"
            profile = admin
            permit
        }
    }
}
```

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

Attribute-related directives are:

- `default attribute = ( permit | deny )`

  This directive specifies whether the daemon is to accept or reject unknown attributes sent by the NAS (default: `deny`).

- (`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.

Numbered Attributes
A `%%d` added to an attribute will will result in a numbered attribute, starting to count at `1` (`%%n` would start counting at `0`). TACCACS+ example:

```
set "route#%%d" = "192.168.0.0 255.255.255.0 10.0.0.1"
set "route#%%d" = "192.168.1.0 255.255.255.0 10.0.0.1"
set "route#%%d" = "192.168.2.0 255.255.255.0 10.0.0.1"
```

results in

```
set "route#1" = "192.168.0.0 255.255.255.0 10.0.0.1"
set "route#2" = "192.168.1.0 255.255.255.0 10.0.0.1"
set "route#3" = "192.168.2.0 255.255.255.0 10.0.0.1"
```

Auto-numbering works for RADIUS, too:

```
set radius[Cisco:Cisco-AVPair] = "ip:inacl#%%d=deny ip any 192.168.0.0 0.0.255.255"
set radius[Cisco:Cisco-AVPair] = "ip:inacl#%%d=permit ip any any"
```

results in

```
set radius[Cisco:Cisco-AVPair] = "ip:inacl#1=deny ip any 192.168.0.0 0.0.255.255"
set radius[Cisco:Cisco-AVPair] = "ip:inacl#2=permit ip any any"
```

Variables
The same variables supported for logging can be used as attribute values, too. Example: `set uid = "${uid}"`

- `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.

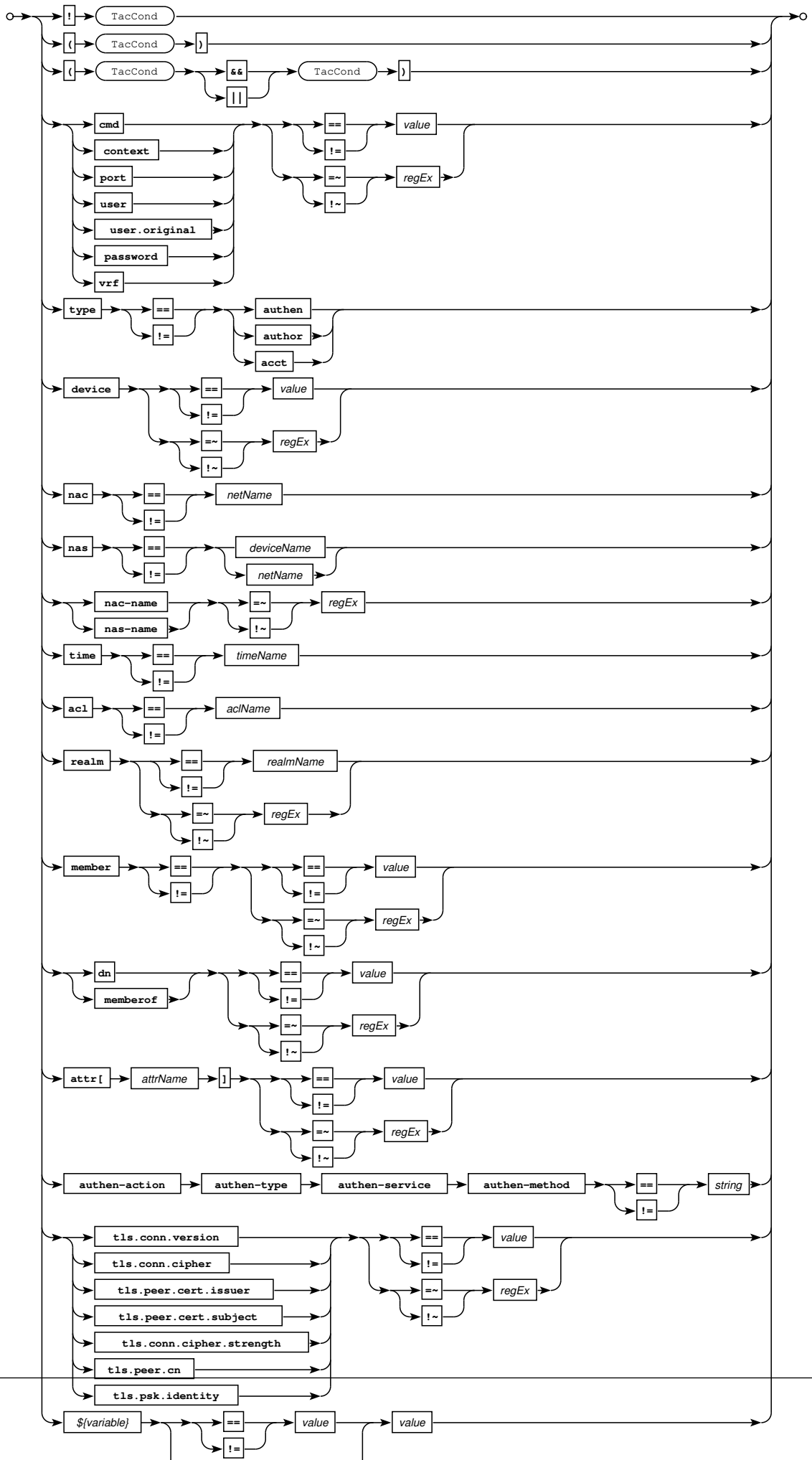
Conditions:

Left-hand side	Operators	Right-hand side	Comment
" *string* "	== != =~ !~	String or REGEX	Log variable substitutions will apply
aaa.protocol.allowed	== !=	one (or more, comma-separated) of radius (covers all RADIUS transports), radius.udp, radius.tcp, radius.dtls, radius.tls, tacacs (covers all TACACS+ transports), tacacs.tcp, tacacs.tls	
acl	== !=	ACL object name	
arg[*attr*]	== != =~ !~	String or REGEX	arg[protocol], arg[service], ...
authen-action	== != =~ !~	String or REGEX	login, chpass
authen-method	== != =~ !~	String or REGEX	login, enable, ppp
authen-service	== != =~ !~	String or REGEX	none, line, enable, local, tacacs+
authen-type	== != =~ !~	String or REGEX	ascii, pap, chap, mschap, mschapv2, sshkey, sshcer
client	== != =~ !~	Net object name or IP address or string.	Net incl. parents
client.name	== !=	Net object name	Client remote address
client.address	== != =~ !~	String or REGEX	TACACS+ client remote address or RADIUS Calling-Station-Id, if available
client.dnsname	== != =~ !~	Client DNS PTR record	
cmd	== != =~ !~	String or REGEX	shell command line
conn.protocol	== !=	Protocol	tcp, udp
conn.transport	== !=	Transport	tls, dtls, tcp, udp
context	== != =~ !~	String or REGEX	current exec context
device	== !=	Device (host) object name, net object name or device address	incl. parents
device.name	== != =~ !~	Device (host) or net object name	incl. parents
device.address	== != =~ !~	Device (host) address	
device.dnsname	== != =~ !~	Device (host) DNS PTR record	
device.tag	== != =~ !~	tagName	
device.tag	== !=	user.tag	False if no match, true else.
dn	== != =~ !~	String or REGEX	MAVIS dn attribute
identity-source	== != =~ !~	String or REGEX	authenticating/authorizing MAVIS module

Left-hand side	Operators	Right-hand side	Comment
`member`	`==` `!=` `=~` `!~`	String or REGEX	group membership
`memberof`	`==` `!=` `=~` `!~`	String or REGEX	MAVIS memberOf attribute
`nac`	`==` `!=` `=~` `!~`	[deprecated, use `client`] String or REGEX	net object name (incl. parents), client remote address (rem_addr)
`nac-name`	`==` `!=` `=~` `!~`	[deprecated, use `client.name`] String or REGEX	client DNS PTR
`nas`	`==` `!=` `=~` `!~`	[deprecated, use `device`] String or REGEX	device or net object name (incl. parents), matches NAC remote address (rem_addr)
`nas-name`	`==` `!=` `=~` `!~`	[deprecated, use `device.name`] String or REGEX	NAS/NAD DNS PTR
`password`	`==` `!=` `=~` `!~`	String or REGEX	session user password
`port`	`==` `!=` `=~` `!~`	[deprecated, use `device.port`] String or REGEX	session port (vty02, console, ...)
`priv-lvlp`	`==` `!=` `=~` `!~`	String or REGEX	session privilege level (0 ... 15) reported by the device
`protocol`	`==` `!=` `=~` `!~`	String or REGEX	session protocol (ppp, ...)
`radius[` *RADIUS attribute* `]`	`==` `!=` `=~` `!~`	RADIUS attribute value	
`realm`	`==` `!=`	Realm object name	
`server.address`	`==` `!=` `=~` `!~`	Server address	
`server.name`	`==` `!=` `=~` `!~`	Server host name	
`server.port`	`==` `!=` `=~` `!~`	Server TCP port	
`service`	`==` `!=` `=~` `!~`	String or REGEX	TACACA+ session service (shell, ...) or RADIUS Service-Type
`time`	`==` `!=`	Time object name	
`tls.conn.cipher`	`==` `!=` `=~` `!~`	String or REGEX	TLS specific data
`tls.conn.cipher.strength`	`==` `!=` `=~` `!~`	String or REGEX	TLS specific data
`tls.conn.version`	`==` `!=` `=~` `!~`	String or REGEX	TLS specific data
`tls.peer.cert.issuer`	`==` `!=` `=~` `!~`	String or REGEX	TLS specific data
`tls.peer.cert.subject`	`==` `!=` `=~` `!~`	String or REGEX	TLS specific data
`tls.peer.cn`	`==` `!=` `=~` `!~`	String or REGEX	TLS specific data
`tls.psk.identity`	`==` `!=` `=~` `!~`	String or REGEX	TLS specific data
`type`	`==` `!=` `=~` `!~`	String or REGEX	authen, author, acct
`user`	`==` `!=` `=~` `!~`	String or REGEX	session user

Left-hand side	Operators	Right-hand side	Comment
`user.tag`	`==` `!=` `=~` `!~`	tagName	
`user.tag`	`==` `!=`	`device.tag`	False if no match, true else.

Railroad diagrams for conditions:


!
TacCond
(
TacCond
)
(
TacCond
&&
||
TacCond
)
cmd
context
port
user
user.original
password
vrf
==
!=
=~
!~
value
regEx
type
==
!=
authen
author
acct
device
==
!=
=~
!~
value
regEx
nac
==
!=
netName
nas
==
!=
deviceName
netName
nac-name
nas-name
=~
!~
regEx
time
==
!=
timeName
acl
==
!=
aclName
realm
==
!=
=~
!~
realmName
regEx
member
==
!=
==
!=
=~
!~
value
regEx
dn
memberof
==
!=
=~
!~
value
regEx
attr[
attrName
]
==
!=
=~
!~
value
regEx
authen-action
authen-type
authen-service
authen-method
==
!=
string
tls.conn.version
tls.conn.cipher
tls.peer.cert.issuer
tls.peer.cert.subject
tls.conn.cipher.strength
tls.peer.cn
tls.psk.identity
==
!=
=~
!~
value
regEx
${variable}
==
!=
value
value

`cmd` and `context` may be used in `shell` context only. `tls_*` conditions require *libtls*.

4.15 Rewriting User Names

A script may refer to a rewrite profile defined at realm level to rewrite user names. For example, the following will map both `admin` and `root` to `jane.doe`, and convert all other usernames to lower-case:

```
rewrite rewriteRule {
    rewrite /^admin$/ jane.doe
    rewrite /^root$/ jane.doe
    rewrite /^.*$/ \L$0
}

device ... {
    ...
    script { rewrite user = rewriteRule }
    ...
}
```

You can limit the usage of a rewritten-to user with the `rewritten-only` directive, e.g.:

```
    rewrite rewriteRule {
        rewrite /^.*$/ nopassword
    }
    user nopassword {
        password login = permit
        password pap = login
        member = ...
        rewritten-only
    }
```

4.16 Users

The basic form of a user declarations is

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

The following declarations are valid in *user* context:

- `alias =` *alternateUserName*

  Implement an alternate user name. This may be used multiple times.

- `password login [ fallback ] = (( clear | crypt | pbkdf2 | asa | 4 | 5 | 6 | 7 | 8 | 9 )` *password* `| mavis | permit | deny | error )`

  The `login` password authenticates `shell` log-ins to the server.

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

  For the argument after `crypt` you may use whatever hashes your *crypt(3)* implementation supports.

  If the `mavis` keyword is used instead, the password will be looked up via the **MAVIS** back-end. 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.

  If you're using `password login = mavis`, the `fallback` password will be used if there's a *MAVIS* backend error.

Fallback to LOCAL
Setting the login password to `error` will cause the session to be rejected with a TACACS+ error status. This will likely instruct your device to fall back to local authentication.

- `password pap [ fallback ] = ((clear|crypt|pbkdf2|asa|4|5|6|7|8|9) password|login|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 hashed, or may be looked up via the **MAVIS** back-end. You can even map pap to login globally by configuring `password pap = login` in realm context.

  If you're using `password pap = mavis`, the `fallback` password will be used if there's a *MAVIS* backend error.

- `password chap =((clear|6) password|permit|deny|mavis)`

  For CHAP authentication, a cleartext password is required. Alternatively, a suitable MAVIS backend can be used.

- `password ms-chap =((clear|6) password|permit|deny|mavis)`

  For MS-CHAP authentication, a cleartext password is required. Alternatively, a suitable MAVIS backend can be used.

- `password{ ... }`

  This directive allows for nested specification of passwords. Example:

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

- `enable [ level ] = (permit|deny|login|(clear|crypt|pbkdf2|asa|4|5|6|7|8|9) password)`

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

- `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.

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

  The user 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 user profile will be invalid after the given date.

- `member =` *groupOne*[`,` *groupTwo*]*

  This specifies group membership. A user can be a member of multiple groups and groups can be members of a parent group.

- `profile =` *profileName*

  `profile {` *profileAttr* }

  `net` *netName* { *netAttr* }

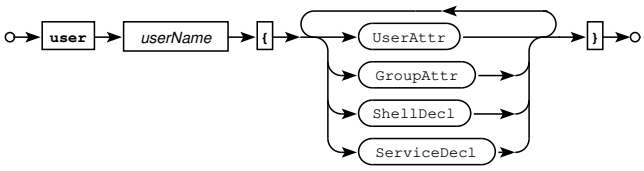
  `time` *timeName* { *timeAttr* }

  A profile may be assigned to a user directly, the main ruleset will be then skipped.

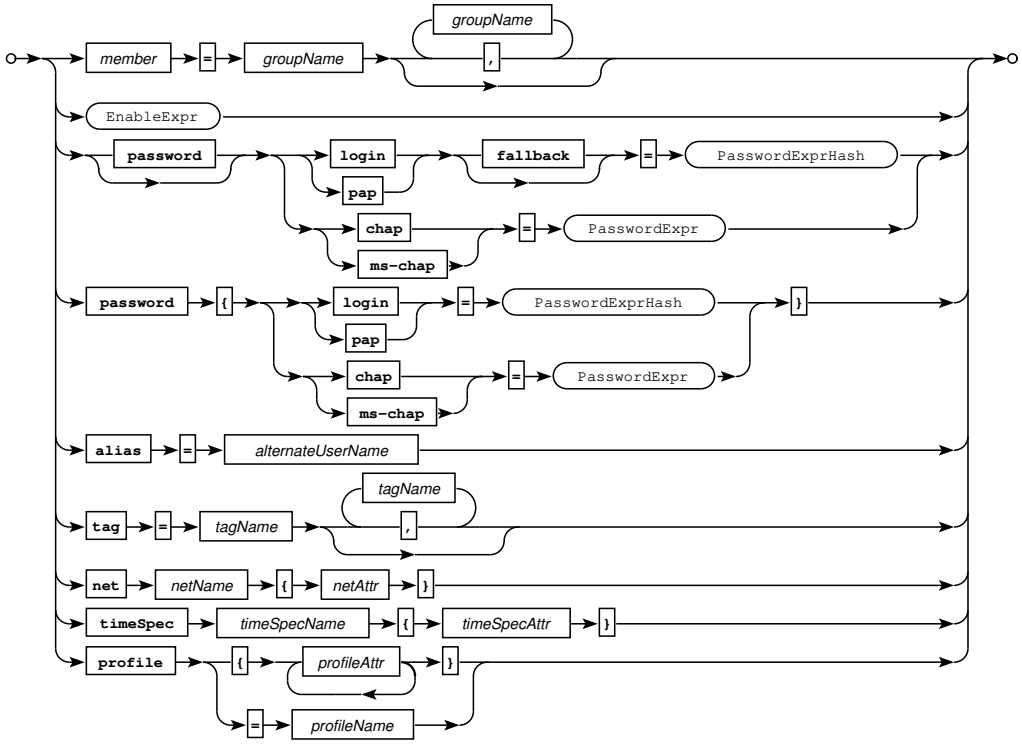
  Also, a profile definition can be part of the user context, and the profile may refer to `net` and `time` definitions defined there:

```
   user demo {
       password login = clear demo
       password pap = login
       net localhost { address = 127.0.0.1 }
time sunday { "* * * * 0" }
       profile {
           acl {
               if (client.name != localhost)
                   deny
               if (time == sunday)
                   deny
   permit
           }
           script {
               if (client.name == localhost)
                   add demoattribute = ignore
               if (service == shell) {
                   if (cmd == "")
                       set priv-lvl = 15
                       permit
               }
           }
        }
   }
```

4.16.1 Railroad Diagrams



Railroad diagram: UserDecl



Railroad diagram: ServiceDecl

4.17 Groups

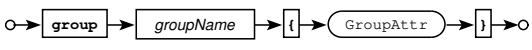
A user can be a member of multiple groups. A user that is a member of a group that comes with a parent group is a member of the latter, too. Group are defined using

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

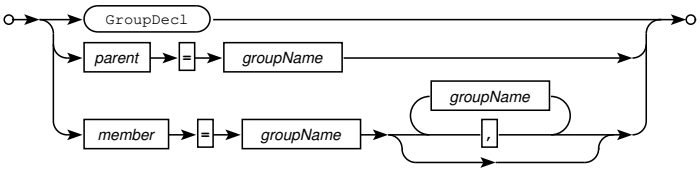
The following key-value pairs are valid for groups:

- `member =` *groupOne*[`,` *groupTwo*]*
  This specifies group membership.
- `parent =` groupName
  The parent of a group can be set explicitly.
- `group` groupName `{` GroupAttr `}`
  Groups may be parents of other groups.

4.17.1 Railroad Diagrams



Railroad diagram: GroupDecl



Railroad diagram: GroupAttr

4.18 Profiles

Profiles are collections of services that can be assigned to users via the policy rule-set. Syntax is

`profile` *profileName* { *profileAttr* }

Also, an unnnamed profile may be configured in user context. This overrides rule evaluation and will be assigned unconditionally, e.g.:

```
user ... {
...
  profile {
    script {
      if (service == shell)
        set priv-lvl = 15
      permit
    }
  }
...
}
```

Assigning an existing profile to an user will also work:

```
user ... {
...
  profile = profileName {
...
}
```

Profiles are collections of services available to a user. A couple of configuration attributes are service specific and only valid in certain contexts:

SHELL (EXEC) Service

Shell startup should have an appropriate script definition

```
script {
    if (service == "shell" && cmd == "")
        permit
}
```

defined. Valid configuration directive within the curly brackets are:

- `script` (= *aclName* | { *tacAction* })

  Service and command authorization is handled using script syntax:

  ```
  script {
      if (service == "shell" && cmd == "")
          permit
      if (cmd =~ /^write term/) deny
      if (cmd =~ /^configure /) deny
      permit
  }
  ```

  As indicated before, `script` will evaluated for authorization requests only.

- `acl` (= *aclName* | { *tacAction* })

  A `acl` allows for refering a script that's being evaluated for both authentication and authorization.

- `profile` *SubProfileName* { ... }

  This defines a new profile that inherits most values from its parent profile.

- `parent =` *ParentProfileName*

  This sets *ParentProfileName* as parent profile. Parent profiles defined in parent realms are accepted, too.

Have a look at the authorization log in case you're unsure what commands and arguments the router actually sends for verification. E.g.,

Non-Shell Services

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

```
script {
    if (service == "ppp" && protocol == "ip") {
        set addr = 1.1.3.4
        permit
    }
}
```

The historical

```
default protocol = permit
```

will no longer be recognized but can be replaced with a simple

```
script {
    permit
}
```

For a Juniper Networks-specific authorization service, use:

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

Likewise, for Raritan Dominion SX IP Console Servers:

```
script {
    if (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:

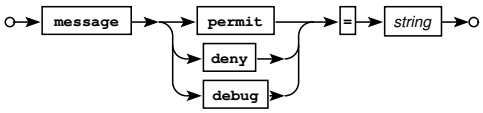
```
set shell:roles = "\"network-admin\""
```

and

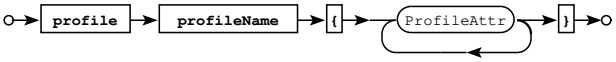
```
set shell:roles = '"network-admin"'
}
```

are equivalent, but the latter is more readable.

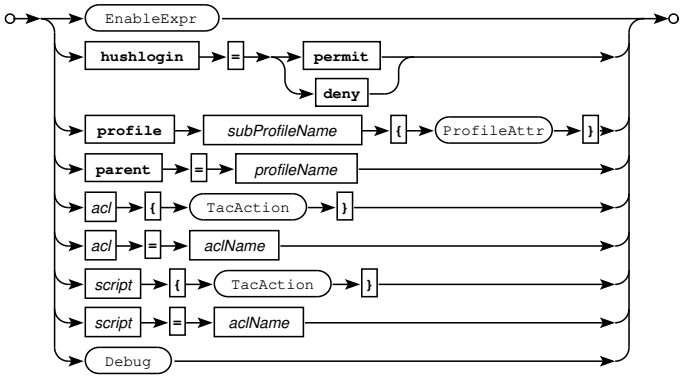
4.19 Railroad Diagrams



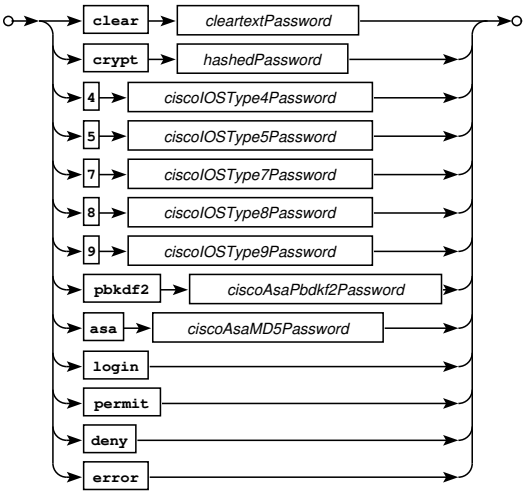
Railroad diagram: UserMessage



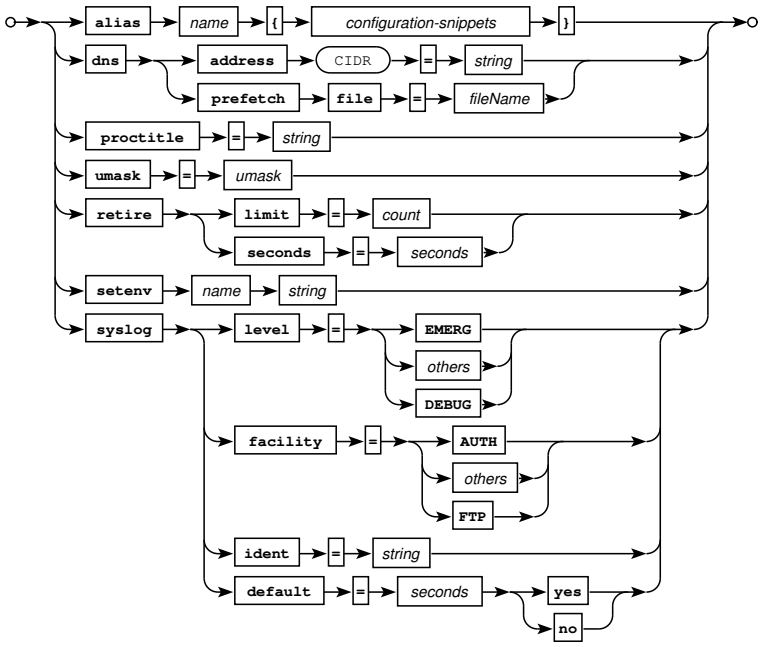
Railroad diagram: ProfileDecl



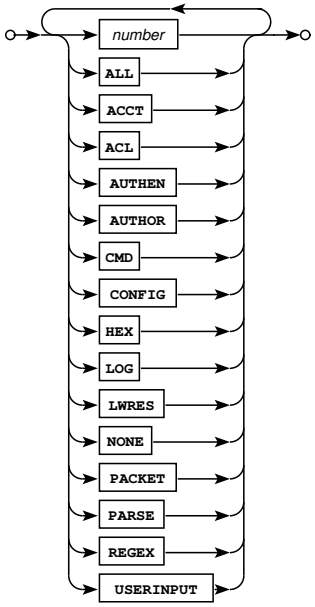
Railroad diagram: ProfileAttr



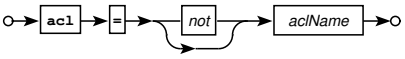
Railroad diagram: PasswordExprHash



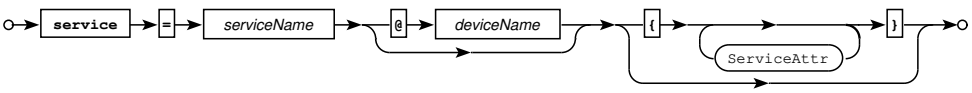
Railroad diagram: TopLevelAttr



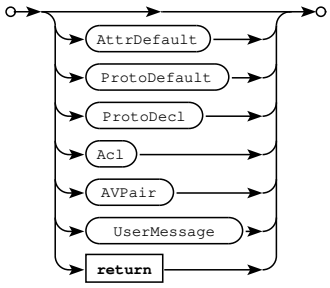
Railroad diagram: Debug



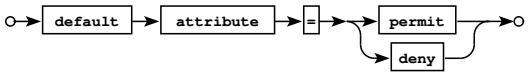
Railroad diagram: Acl



Railroad diagram: ServiceDecl



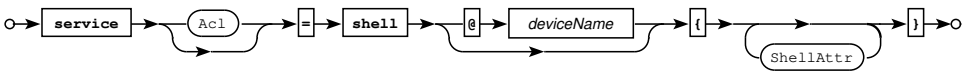
Railroad diagram: ServiceAttr



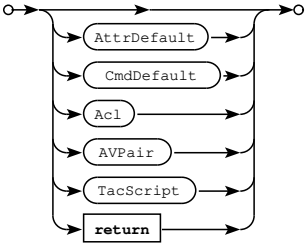
Railroad diagram: AttrDefault



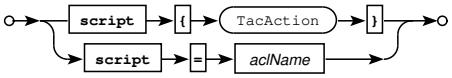
Railroad diagram: AVPair



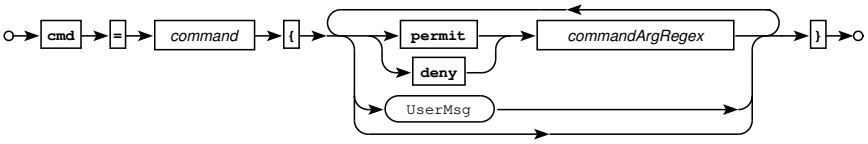
Railroad diagram: ShellDecl



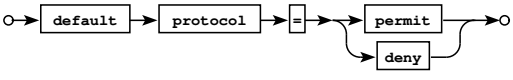
Railroad diagram: ShellAttr



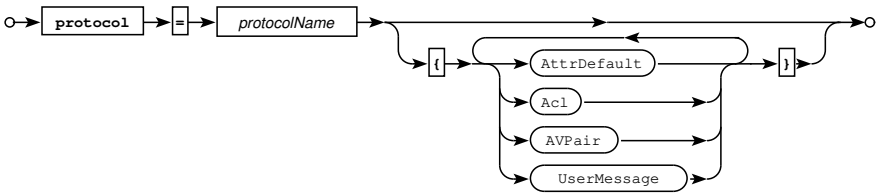
Railroad diagram: TacScript



Railroad diagram: ShellCommandDecl



Railroad diagram: ProtoDefault



Railroad diagram: ProtoDecl

4.20 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 back-ends 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.

4.21 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.

4.22 Configuring User Authentication

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

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

```
user mary {
    password chap = clear "chap password"
    password pap  = clear "inbound pap password"
    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 back-end. You should consider using this option in conjuction with the more sophisticated back-ends (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 back-end,

```
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 back-end (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 back-end, 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 }
```

4.23 Configuring Expiry Dates

An entry of the form:

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

will cause the user 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.

4.24 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.

4.25 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.

4.26 Authorizing Commands

Exec commands are those commands which are typed at a NAS 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 telling the ruleset to apply a profile to the user. See the Profile section for details.

4.27 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.

4.28 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.

4.29 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.

4.29.1 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).

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-ng_ldap.pl` is an authentication/authorization back-end for the *external* module. It interfaces to various kinds of LDAP servers, e.g. OpenLDAP, Fedora DS and Active Directory. The server type is detected automatically. Its behaviour is controlled by a list of environmental variables:

Variable	Description
`LDAP_HOSTS`	Space-separated list of LDAP URLs or IP addresses or device names Examples: `"ldap01 ldap02"`, `"ldaps://ads01:636 ldaps://ads02:636"`
`LDAP_SCOPE`	LDAP search scope for users (`base`, `one`, `sub`) Default: `sub`
`LDAP_SCOPE_GROUP`	LDAP search scope for groups (`base`, `one`, `sub`) Default: `sub`
`LDAP_BASE`	Base user search DN of your LDAP server Example: `dc=example,dc=com`
`LDAP_BASE_GROUP`	Base group search DN of your LDAP server Example: `dc=example,dc=com`
`LDAP_CONNECT_TIMEOUT`	Timeout for initital connect to remote LDAP server. Default: 1 (second).

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

`LDAP_FILTER_GROUP`	LDAP search filter for groups. Default: `"(&(objectclass=groupOfNames)(member=%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
`LDAP_MEMBEROF_REGEX`	Regular expression to derive group names from memberOf, Default: `"^cn=([^,]+),.*"`
`LDAP_TACMEMBER`	LDAP attribute to use for group membership (fallback only). Default: `"tacMember"`
`LDAP_TACMEMBER_MAP_OU`	Map organizational units from user DN to group membership. Default: unset
`USE_STARTTLS`	If set, the server is required to support `start_tls`. Default: unset
`FLAG_AUTHORIZE_ONLY`	Don't attempt to authenticate users. Default: unset

Variable	Description
TLS_OPTIONS	Extra options for use with LDAPS or start_tls, in Perl hash syntax. See the Net::LDLAP documentation for details. The back-end checks the certificate of the server (`verify => 'require'`), unless TLS_OPTIONS sets `verify`. Name the CA of the server with `cafile`. `verify => 'none'` turns the check off. Default: unset (the certificate of the server is checked) Example: `"sslversion => 'tlsv1_3'"`, `"cafile => '/path/to/ca.pem'"`
FLAG_AUTHORIZE_ONLY	Don't attempt to authenticate users. Default: unset
LDAP_NESTED_GROUP_DEPTH	Limit nested group lookups to the given value. Unlimited if unset. Example: 1

5.1.1 Multi-threaded LDAP Backend

`ldapmavis-mt` (*mt* stands for *multi-threaded*) basically evaluates the same envrionment variables as `mavis_tacplus-ng_ldap.p`
It needs to be be invoked via the `external-mt` module and is suited for long-lasting authentication session, e.g. due to multi-factor authentication.

`ldapmavis-mt` does not read `TLS_OPTIONS`. It sets the TLS options of libldap from environment variables of the same names, e.g. `LDAP_OPT_X_TLS_CACERTFILE` or `LDAP_OPT_X_TLS_PROTOCOL_MIN`; see the `ldap_set_option` manual page and `ldapmavis-mt -h` for the list. The back-end checks the certificate of the server: `LDAP_OPT_X_TLS_REQUIRE_C` defaults to `LDAP_OPT_X_TLS_DEMAND`. Set `LDAP_OPT_X_TLS_CACERTFILE` to the CA of the server, or set `LDAP_OPT_X_TLS_` to `LDAP_OPT_X_TLS_NEVER` to turn the check off.

```
setenv LDAP_OPT_X_TLS_CACERTFILE = "/path/to/ca.pem"
```

5.2 PAM back-end

Example configuration for using *Pluggable Authentication Modules*:

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

id = tac_plus {
  mavis module groups {
    resolve gids = yes
    resolve gids attribute = TACMEMBER
    groups filter = /^(guest|staff)$/
  }
  mavis module external {
    exec = /usr/local/sbin/pammavis "pammavis" "-s" "sshd"
  }
  user backend = mavis
  login backend = mavis
  device global { address = 0.0.0.0/0 key = demo }

  profile staff {
    service shell {
      script {
          if (cmd == "") {
            set priv-lvl = 15
            permit
          }
    }
  }
  profile guest {
    service shell {
```

```
      script {
          set priv-lvl = 15
          if (cmd =~ ^/show /)
            permit
          deny
      }
    }
  }
}
```

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.

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
      #
      # There are more modern password hashes available via mkpasswd:
      # setenv MKPASSWD=/usr/bin/mkpasswd
      # setenv MKPASSWDMETHOD=yescrypt
      #
      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 devices 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 back-end.

5.7 Error Handling

If a back-end 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 back-end error:

```
...
# set the time interval you want the user to be valid if the back-end 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:

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

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

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

Backend failures are detected per-process. However, you can sync fallback state between worker processes via the file system, on a per-realm basis:

```
authentication fallback file = /tmp/tac/my_realm/fallack
```

6 RADIUS

tac_plus-ng comes with RADIUS support: PAP/CHAP/MSCHAPv1/MSCHAPv2 authentications, accounting and Downloadable ACLs are supported. This should be sufficient for handling administrative access, and it obviously provides a common configuration for both *TACACS+* and *RADIUS*

If you don't care about *RADIUS*: That functionality will not be available unless a *RADIUS dictionary* (see below) is configured, and TLS/DTLS/Message-Signing absolutely requires building with *OpenSSL* support.

Also, you can selectively disable the protocols you want to be supported on a per-realm basis:

```
aaa.protocol.allowed = tacacs,tacacs.tcp,tacacs.tls,radius,radius.udp,radius.tls,radius. ↩
   dtls
 # Actually "tacacs" covers both "tacacs.tcp" (plain TACACS+) and "tacacs.tls" (Secure  ↩
    TACACS+").
 # Likewise, "radius" covers RADIUS, RADIUS-TCP, RADSEC and RADIUS-DTLS.
```

6.1 Accepting RADIUS Queries

By default, **spawnd** will only care for TCP sessions. This is sufficient for *RADIUS-over-TLS (RADSEC)*, but plain legacy RADIUS requires UDP. This can easily be enables in a **spawnd** directive:

```
id = spawnd {
        background = no
        listen { port = 49 } # TACACS+
        listen { port = 1812 protocol = UDP } # RADIUS Access
        listen { port = 1813 protocol = UDP flag = accounting } # RADIUS Accounting
}
```

In fact, the daemon auto-recognizes the supported protocols. You can use the very same TCP port for *TACACS+*, *TACACS+-over-TLS* and *RADSEC-over-TLS*, and the UDP port(s) specified for *RADIUS* will accept both authentication and accounting packets. However, defining the secondary port (with the `accounting` flag enabled is required for correct handling of RADIUS Server-Status requests.

UDP RADIUS sessions will be accepted only if a matching key (shared secret) exits. You can define that in `host` context:

```
    radius.key = mysecretkey
```

RADSEC doesn't need this key definition.

6.2 RADIUS DACLs

CISCO downloadable ACLs (dACLs, DACLs) are supported for RADIUS. Downloadable ACL sample definition:

```
    dacl demoacl {
        # prefix = "ip:inacl" # This is the default.
        # version = 12345678  # This is explained somewhere below.
        data = "
            deny ip host 10.0.0.0 any
            permit ip any any
        "
    }
```

In user profile context, you can then

```
    set radius[Cisco:Cisco-AVPair] = "ACS:CiscoSecure-Defined-ACL=${dacl:demoacl}"
```

which will instruct the device to query for that ACL via RADIUS.

The device will see the `demoacl` portion as something like `#ACSACL#demoacl-00bc614e`, where the suffix is, in hexadecimal, either the manually set version number (`version = 12345678`, that is 0x00bc614e) or the first 4 bytes of the ACL MD5 checksum. The version is always printed as 8 hex digits, never in decimal.

MAVIS support is there, too. One of the demo scripts (`mavis_tacplus-ng-demo-database.pl`) shows how to handle that.

No syntax checking is done for the `data` argument. If you mess up on that then the device will likely reject the connection.

6.3 RADIUS Dictionary

There's no file format standard for RADIUS dictionaries, so **tac_plus-ng** comes with its own, too. Here's an excerpt from `tac_ls-ng/sample/radius-dict.cfg` which demonstrates the syntax:

```
radius.dictionary {
        attribute User-Name             1       string
        attribute User-Password         2       string
        attribute CHAP-Password         3       octets
```

```
        attribute NAS-IP-Address        4       ipaddr
        attribute NAS-Port              5       integer
        attribute Framed-IP-Address     8       ipaddr
        attribute Framed-IP-Netmask     9       ipaddr
        attribute Class                 25      octets

        attribute NAS-Port-Type         61      integer
        {
                Async                   0
                Sync                    1
                ISDN                    2
                ISDN-V120               3
                ISDN-V110               4
                Virtual                 5
                Cable                   17
...
                Wireless-Other          18
                Wireless-802.11         19
        }

        attribute Service-Type 6 integer
        {
                Login-User              1
                Framed-User             2
                Callback-Login-User     3
...
                Authorize-Only          17
                Framed-Management       18
        }

        attribute Login-IP-Host         14      ipaddr
        attribute Login-Service         15      integer
...
        attribute Framed-IPv6-Pool      100     string
        attribute Framed-IPv6-Address   168     ipv6addr

        attribute Acct-Session-Time     46      integer
        attribute Acct-Input-Packets    47      integer
        attribute Acct-Output-Packets   48      integer

        attribute Acct-Terminate-Cause  49      integer
        {
                User-Request            1
                Lost-Carrier            2
                Lost-Service            3
...
                User-Error              17
                Host-Request            18
        }

        attribute Acct-Multi-Session-Id 50      string
        attribute Acct-Link-Count       51      integer
}

radius.dictionary cisco 9 {
        attribute Cisco-AVPair 1 string
}
```

You can put this dictionary data directly into your main configuration file or put it somewhere else and just `include` it, which is recommended.

Multi-byte attribute (and length) dictionaries are supported, too:

```
radius.dictionary vendor-name vendor-id format=attribute bytes,length bytes
```

(This is similar to FreeRADIUS, but there are quite certainly some missing options.)

6.4 RADIUS Rule Set

RADIUS attributes are accessible via `radius[`*AttributeName*`]` and can be used both in conditions and `set` clauses:

```
    if (aaa.protocol == radius) {
        if (radius[Service-Type] ==  Administrative-User)  {
            set radius[cisco:Cisco-AVPair] = "shell:priv-lvl=15"
            permit
        }
        set radius[cisco:Cisco-AVPair] = "shell:priv-lvl=7"
        permit
    }
```

6.5 RADIUS Logging

TACACS+ and *RADIUS* share the `connection log`. Dedicated *RADIUS* logs for authentication and acounting can be enabled:

```
        log accesslog { destination = /tmp/tac/access.log }     # TACACS+
        log authorlog { destination = /tmp/tac/author.log }     # TACACS+
        log acctlog { destination = /tmp/tac/acct.log }         # TACACS+
        log rad-accesslog { destination = /tmp/rad/access.log } # RADIUS
        log rad-acctlog { destination = /tmp/rad/acct.log }     # RADIUS
        log connlog { destination = /tmp/conn.log }             # shared

        access log = accesslog
        authorization log = authorlog
        accounting log = acctlog
        radius.access log = rad-accesslog
        radius.accounting log = rad-acctlog
        connection log = connlog
```

6.6 RADIUS Example Configurations

Sample configurations are available in the `tac_plus-ng/samples/` directory.

7 Debugging

7.1 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.

7.2 Trace Options

Trace (or debugging) options may be specified in *global*, *device*, *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.

8 Frequently Asked Questions

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

  No, unless your favourite text editor does qualify.

- **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#
```

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

  CHAP requires 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, not just the whim of some Cisco engineer who drank too much coffee late one night.

  If we encrypted the CHAP passwords in the database, then we'd need to keep a key around so that the server can decrypt them when CHAP 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 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 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 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

- **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 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.

```
profile ... {
    ...
    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.

- **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.

- **What about MSCHAP?**

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

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

  Marc Huber thinks that MSCHAP relevance is less than zero and expects it to be removed from the standard, as nobody uses it anyway.

9 Multi-tenant setups

While using a dedictated tac_plus-ng installation per tenant is certainly possible it lacks some elegance. There are other ways:

A single daemon can tell tenants apart by

- device identity, either
  - by NAD IP address or
  - by certificate common name (currently irrelevant, as there's no NAD support)
- realms, which are determined
  - by tacacs+ destination port or
  - by VRF (Linux, mostly)

Using the IP-based device identity should be sufficient for simple setups, but these don't scale and don't handle IP address conflicts. Options to cope with the latter involve *realms*. A realm is most basicly a text string the tcp listener (spawnd) assigns to a connection based on TCP destination port:

```
id = spawnd {
    ...
    listen { port = 49001 realm = customer1 }
    listen { port = 49002 realm = customer2 }
    ...
}
```

In case VRFs aren't an option you can use HAProxy instances to transparently relay TACACS+ connections to tac_plus-ng:

```
id = spawnd {
    ...
    listen { port = 49001 realm = customer1 haproxy = yes }
    listen { port = 49002 realm = customer2 haproxy = yes }
    ....
}
```

tac_plus-ng will then take the NAD IP from the HAProxy protocol v2 header.

Otherwise, the "listen" directive can be limite to your locally defined VRFs:

```
id = spawnd {
    ...
    listen { port = 49000 realm = customer1 vrf = blue }
    listen { port = 49000 realm = customer2 vrf = red }
    ...
}
```

On Linux, if you set `net.ipv4.tcp_l3mdev_accept=1`, you can even get away with

```
id = spawnd { ... listen { port = 49000 } ... }
```

and the daemon will use the VRF name your clients did connect from as realm name.

9.1 AD, Realms and Tenants

The suggested setup for giving customers limited access to NADs is:

```
id = tac_plus-ng {
    mavis module external { your AD configuration goes here }

    profile ... { ... }

    realm customer1 {

        net custsrc { the IP ranges the end customer may log in from  }

        rewrite normalizeCustomerAccount {
           rewrite /^.*$/ cust1-\L$0
        }

        net custnet { the IP ranges the end customer may log in from  }

        device customer1 {
            ....
            script {  if (nac == custsrc) rewrite user = normalizeCustomerAccount }
        }

        ruleset {
            rule customer {
                if (nac == custnet) {
                   if (member == ...) { profile = ... permit }
                   deny
                }
               if (member == ...) profile = ... permit
               deny
        }
}
```

In this example, you can easily share your LDAP (or AD) server between your own admin users and multiple tenants. The daemon will automatically prefix the customer accounts with a prefix and convert them to lower case. Note that the username rewriting happens using a script in device context. Rewriting won't work in scripts anywhere else.

10 AAA rule tracing

The distribution includes the `tactrace.pl` Perl script. It's usually not installed automatically, due to some Perl dependencies that need to be met. It requires a couple of CPAN Perl modules, and a custom one. Your OS distribution might provide re-built packages for `Net::IP` and/or `Net::TacacsPlus::Packet`, so check for this first. For unavailable packages, you can do a manual install using cpan. Example for Ubuntu:

```
sudo apt install libnet-ip-perl
sudo cpan install Net::TacacsPlus::Packet
```

Then `cd` to `tac_plus-ng/perl` and run `make`. `tactrace.pl` should now be ready to use.

Usage information:

```
$ This is a TACACS+ and RADIUS AAA validator for tac_plus-ng.

Usage: ./tactrace.pl [ <Options> ] [ <attributes> ... ]

attributes are authorization or accounting AV pairs, default is:
        "service=shell" "cmd*"

Options:
  --help                show this text
  --defaults=<file>     read default settings from <file>
```

```
  --mode=<mode>         authc, authz or acct [authz]
  --username=<username> username [username]
  --password=<password> user password [$TACTRACEPASSWORD]
  --port=<port>         port [vty0]
  --remote=<client ip>  remote client ip [127.0.0.1]
  --key=<key>           encryption key [demo]
  --realm=<realm>       realm [default]
  --nad=<address>       NAD (router/switch/...) IP address [127.0.0.1]
  --authentype=<type>   authen_type [ascii]
  --authenmethod=<n>    authen_method [tacacsplus]
  --authenservice=<n>   authen_method [login]
  --exec=<path>         executable path [/usr/local/sbin/tac_plus-ng]
  --conf=<config>       configuration file [/usr/local/etc/tac_plus-ng.cfg]
  --id=<id>             id for configuration selection [tac_plus-ng]
  --radius                 test RADIUS (RADIUS/TCP, actually) instead of TACACS+
  --radius-dict=<file>  radius dictionary to use [limited built-in dictionary]

For authc the password can be set either via the environment variable
TACTRACEPASSWORD or the defaults file. Setting it via a CLI option isn't
supported as the password would show up as clear text in the process listing.
```

Example:

```
# tactrace.pl --conf extra/tac_plus-ng.cfg-ads --user user01
127.0.0.1 ---<start packet>---
127.0.0.1 session id: 00000001, data length: 46
127.0.0.1 AUTHOR, priv_lvl=0
127.0.0.1 authen_type=ascii (1)
127.0.0.1 authen_method=tacacs+ (6)
127.0.0.1 service=login (1)
127.0.0.1 user_len=6 port_len=4 rem_addr_len=9 arg_cnt=2
127.0.0.1 user (len: 6): user01
127.0.0.1 0000 75 73 65 72 30 31                                  user01
127.0.0.1 port (len: 4): vty0
127.0.0.1 0000 76 74 79 30                                        vty0
127.0.0.1 rem_addr (len: 9): 127.0.0.1
127.0.0.1 0000 31 32 37 2e 30 2e 30 2e  31                        127.0.0. 1
127.0.0.1 arg[0] (len: 13): service=shell
127.0.0.1 0000 73 65 72 76 69 63 65 3d  73 68 65 6c 6c            service= shell
127.0.0.1 arg[1] (len: 4): cmd*
127.0.0.1 0000 63 6d 64 2a                                        cmd*
127.0.0.1 ---<end packet>---
127.0.0.1 Start authorization request
127.0.0.1 looking for user user01 in MAVIS backend
127.0.0.1 user found by MAVIS backend, av pairs:
  MEMBEROF            "CN=tacacs_admins,OU=Groups,DC=example,DC=local","CN=tacacs_readwrite ↩
      ,OU=Groups,DC=example,DC=local"
  USER                user01
  DN                  CN=user01,CN=Users,DC=example,DC=local
  IPADDR              127.0.0.1
  SERVERIP            127.0.0.1
  REALM               default
  TACMEMBER           "admins"
127.0.0.1 verdict for user user01 is ACK
127.0.0.1 user 'user01' found
127.0.0.1 evaluating ACL default#0
127.0.0.1 pcre2: '^CN=tacacs_admins,' <=> 'CN=tacacs_admins,OU=Groups,DC=example,DC=local'  ↩
    = 1
127.0.0.1  line 79: [memberof] <pcre-regex> '^CN=tacacs_admins,' => true
127.0.0.1  line 79: [profile] 'admins'
127.0.0.1  line 79: [permit]
127.0.0.1 ACL default#0: match
```

```
127.0.0.1 user01@127.0.0.1: ACL default#0: permit (profile: admins)
127.0.0.1  line 45: [service] = 'shell' => true
127.0.0.1  line 47: [cmd] = '' => true
127.0.0.1  line 47: [set] 'priv-lvl=15'
127.0.0.1  line 48: [permit]
127.0.0.1 nas:service=shell (passed thru)
127.0.0.1 nas:cmd* (passed thru)
127.0.0.1 nas:absent srv:priv-lvl=15 -> add priv-lvl=15 (k)
127.0.0.1 added 1 args
127.0.0.1 Writing AUTHOR/PASS_ADD size=30
127.0.0.1 ---<start packet>---
127.0.0.1 session id: 00000001, data length: 18
127.0.0.1 AUTHOR/REPLY, status=1 (AUTHOR/PASS_ADD)
127.0.0.1 msg_len=0, data_len=0, arg_cnt=1
127.0.0.1 msg (len: 0):
127.0.0.1 data (len: 0):
127.0.0.1 arg[0] (len: 11): priv-lvl=15
127.0.0.1 0000 70 72 69 76 2d 6c 76 6c  3d 31 35                priv-lvl =15
127.0.0.1 ---<end packet>---
```

11 Bugs

- 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:

     This product includes software developed by Marc Huber (Marc.Huber@web.de).

     Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

  THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ITS AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.