Radiator Policy Server reference guide

Table of Contents

- Servers
  - RADIUS
    - Listen
      - Protocol
      - Port
      - Ip
      - Buffer
    - Policy
    - Clients
- RADIUS client
  - Source
    - Ip
    - Protocol
  - Secret
  - Require_message_authenticator
  - Timeout
- Backends
  - File backend
  - SQL backend
  - LDAP backend
  - RADIUS backend
- Policy
  - Policy conditions
  - Handler
    - Authentication
    - Authorization
    - Accounting
    - Handler statements
    - Handler directives
- Logging
  - Logging sources
    - AAA logging
    - Application logging
  - Logging destinations
  - File
  - Console

    - Memory
    - Common logging attributes
    - Syslog
- Extensible Authentication Protocol (EAP)
    - EAP configuration
    - EAP-MD5-Challenge
    - PEAP
    - TEAP
    - EAP-TLS
    - EAP-TTLS
    - EAP-MSCHAP-V2
- License
- Dictionary
- Capture
    - Capture destinations
- Store
- Supported namespaces, attributes, and data types
    - Data types
    - Format string and filters
    - Namespaces

Servers

One protocol and port per server supported. Each server clause must be configured with a string that represents its name.

Example configuration of a servers clause with two simple RADIUS server configurations:

```
servers {
    radius "server-radius-auth-udp-1812" {
        capture "console-capture";
        listen {...}

        # Allow connections/requests from clients in a client list "clients-radius-all"
        clients "clients-radius-all";

        # (Optional) Handle all requests by AAA policy named "auth"
        #policy "auth";
    }

    radius "server-radius-acct-udp-1813" {
        listen {...}

        # Allow connections/requests from clients in a client list "clients-radius-all"
        clients "clients-radius-all";
    }
}
```

RADIUS

Beginning of a RADIUS server clause. The clause has a name which is defined as a string. Multiple RADIUS clauses can be configured inside the servers clause.

Listen

The listen attribute is used to configure the network interfaces and ports on which the RADIUS server will accept requests. This is used to define specific IP addresses and ports for handling RADIUS traffic.

Example configuration of a listen clause:

```
listen {
    # Transport protocol
    protocol udp;

    # Transport protocol port to listen to
    port 1813;

    # IPv4 address to listen to
    ip 127.0.0.1;
}
```

Protocol

The protocol attribute specifies the type of protocol used by the RADIUS server to receive and handle requests. The parameter type for the protocol attribute is string and the possible values are udp, tcp, and tls.

Port

Specifies which port(s) Radiator will listen on for RADIUS requests. The port parameter is defined as a NonZeroU16 which is a non-zero 16-bit unsigned integer. There can only be one port specified per a listen clause.

Ip

This specifies the ip address(es) on which the server will listen for incoming connections. The ip attribute can be defined as an IPv4 or an IPv6 address and does not need an additional prefix to determine the type.

Recommended listen configuration:

```
listen {
    # Bind to all IPv4 addresses
    ip 0.0.0.0;

    # Bind to all IPv6 addresses
    ip 0::0;
}
```

When the IP attribute is defined as `0.0.0.0` (for IPv4) or `0::0` (for IPv6), the server will listen on all available network interfaces for that particular protocol. If the server is configured with either of those, no other IP addresses with the same protocol can be declared.

Multiple specific IP addresses can also be configured:

```
listen {
    ...

    ip 127.0.0.1;
    ip 192.0.2.42;
    ip 2001:db8::42;
    ip 2001:db8::43;
}
```

Buffer

This optional attribute is used to receive and send buffer size in bytes. The buffer can be defined as a NonZeroU32, which is a non-zero 32-bit unsigned integer. If the buffer field is left unspecified, it will default to the operating system default value.

Policy

This optional attribute is used to handle all requests by AAA policy that is named as the same as the policy name given. The policy attribute works as follows:

1. Check if there exists policy that matches the name, if yes, use it regardless what might be the conditions for the policy.
2. If there is no policy with exact name match, use the first policy that has matching conditions.
3. Use DEFAULT policy if such exists. We always recommend having a DEFAULT policy where authentications can be rejected and accounting accepted.
4. If no matching policy is found, the request is not processed.

The parameter type for the policy is string. For more information, see policy.

Clients

List of clients the RADIUS server will listen to. The client list name is defined as a string. For more information, see RADIUS client.

RADIUS client

A Client clause specifies a RADIUS client that this server will listen to. The client clause must be defined with a string that represents its name. Requests received from any client not named in a Client clause in the configuration file will be silently ignored. The DEFAULT client (if defined) will handle requests from clients that are not defined elsewhere.

You must have a Client clause for every RADIUS client which your server is expected to serve, or else a DEFAULT Client.

Example configuration of a client clause:

```
client "localhost" {
    source {
```

```
        ip 127.0.0.1;
    }
    secret "mysecret";
}
```

Source

This clause contains one or more ip addresses.

Ip

This defines the source address for incoming RADIUS packets. The ip attribute can be defined as an IPv4 or an IPv6 address and does not need an additional prefix to determine the type.

Multiple ip addresses can be defined.

Protocol

This optional attribute specifies the type of transport layer protocol that the client will use to send RADIUS requests. The possible values are udp, tcp, and tls.

Secret

This defines the shared secret that is used to encrypt and decrypt User-Password and some other less frequently used attributes. Shared secret is also used for RADIUS message integrity checking with the exception of Access-Request messages. You must define a shared secret for each Client, and it must match the secret configured into the client RADIUS software. There is no default. The secret can be any number of ASCII characters. Any ASCII character except newline is permitted, but it might be easier if you restrict yourself to the printable characters. For a reasonable level of security, the secret should be at least 16 characters, and a mixture of upper and lower case, digits and punctuation. You should not use just a single recognizable word. The secret is defined as a string.

Require_message_authenticator

This is an optional attribute that indicates whether the RADIUS client should require a message authenticator. The attribute is defined as a boolean. If this attribute is not explicitly defined in the configuration, it will default to 'false'.

Timeout

This optional parameter specifies the amount of time, in seconds, that the client will wait for a response from the RADIUS server before considering the request to have timed out. The timeout is defined as a u32, which is a 32-bit unsigned integer.

Backends

The backends clause tells where end user information is stored. The supported backend types are presented below. Each backend should be configured with a string representing the backend's name. Backends are configured within a `backends` clause:

```
backends {
    # backends are configured here
    file "EXAMPLE_FILE_BACKEND" {...}
}
```

File backend

The file backend allows Radiator to authenticate users against a simple text file. This is useful for basic authentication scenarios and testing purposes. The file contains a list of usernames and passwords, which are used to verify the user's credentials.

Example configuration of a file backend:

```
# file backend configuration
file "USERS_INTERNAL_FILE" {
    filename "/var/lib/radiator/db/users-internal/users-internal.file";
}
```

The filename attribute should be the name of the file backend.

SQL backend

The SQL backend allows Radiator to authenticate and authorize users against various SQL databases, such as MySQL/MariaDB, PostgreSQL, and SQLite. The SQL backends are configured with a name that is defined as a string.

Example configuration of a SQLite backend:

```
sqlite "SQL_DATABASE_USERS_INTERNAL" {
    # Database URL
    # url "sqlite:users-internal.sqlite";

    # Alternatively, configure a database filename
    filename "/var/lib/radiator/db/users-internal/users-internal.sqlite";

    # SQL query named "FIND_USER"
    query "FIND_USER" {
        # SQL statement
        statement "SELECT USERID, PASSWORD FROM USERIDS WHERE USERID = ?";

        # Query argument binding in order
        bindings {
            aaa.identity;
        }

        # Result value mapping
        mapping {
            user.username = "USERID";
            user.password = "PASSWORD";
        }
    }
```

```
    # SQL query named "USER_GROUPS"
    query "USER_GROUPS" {
        # SQL statement
        statement "SELECT groupname FROM groups INNER JOIN group_memberships ON groups.id = gro

        # Query argument binding in order
        bindings {
            aaa.identity;
        }

        # Result values mapping
        mapping {
            user.group += "groupname";
        }
    }
}
```

Query

This clause defines an SQL query operation that retrieves data from the database. The query clause is configured with a name which is defined as a string.

URL

This attribute specifies the connection string used to establish a connection with the SQL database. The format of the URL depends on the specific SQL backend type being used. The url parameter is defined as a string.

LDAP backend

The ldap backend allows Radiator to authenticate and authorize users against an LDAP directory.

Example configuration of an LDAP backend:

```
ldap "ldap.forumsys.com" {
    # LDAP server
    server "ldap.forumsys.com" {
        # LDAP URL
        url "ldap://ldap.forumsys.com:389/";

        # Operation timeout in milliseconds
        timeout 3000;

        # How many sockets/connections at maximum to open
        #connections 10;

        # (Optional) Authentication
        authentication {...}

        # (Optional) TLS client configuration
        #tls {...}
    }

    # A single backend can have multiple LDAP servers configured
```

```
    #server "ldap2" {
    #   ...
    #}

    # (Optional) Per search/operation authentication
    #authentication {
    #    # Basic bind authentication
    #    # Both dn and password parameter support %{...} attribute templates
    #    dn "uid=%{aaa.identity},dc=example,dc=com";
    #    password "%{auth.response}";
    #}

    # LDAP operations

    # LDAP search operation named "user_groups"
    search "user_groups" {
        base "dc=example,dc=com";
        scope sub;
        filter "(&(objectClass=groupOfUniqueNames)(uniqueMember=uid=%{aaa.identity},dc=example,

        # Result values mapping
        mapping {
            user.group += ou;
        }
    }
}
```

Server

This clause defines the LDAP server to connect to. A single backend can have multiple LDAP servers configured. The server is named with a string.

Connections

This attribute specifies the maximum number of sockets/connections that Radiator will open to the LDAP server. This limits the number of concurrent LDAP requests that can be handled, helping to prevent resource exhaustion. The connections parameter is defined as a u16, meaning it's an unsigned 16-bit integer.

Timeout

This attribute specifies the maximum amount of time, in milliseconds, that Radiator will wait for an LDAP operation to complete. This parameter is defined as a u32, which is an 32-bit unsigned integer.

URL

This is the URL of the LDAP server. The URL parameter is defined as a string.

Authentication

The optional authentication clause is used to specify how Radiator authenticates to the LDAP server.

Example configuration of an authentication clause:

```
authentication {
    # Basic bind authentication
    dn "cn=read-only-admin,dc=example,dc=com";
    password "password";

    # SASL External authentication (for example TLS client certificate)
    #external;

    # SASL GSS-API authentication;
    #gss-api;
}
```

Search

Defines an LDAP search operation. This operation is used to query the LDAP directory for user information based on specified criteria.

Example configuration of a search clause:

```
search "find_user" {
    base "uid=%{aaa.identity},dc=example,dc=com";
    scope base;
    filter "objectClass=inetOrgPerson";

    # Result value mapping
    mapping {
        user.username = uid;
        user.password = mail;
    }
}
```

TLS

This optional clause configures TLS (Transport Layer Security) settings for the LDAP connection.

Example configuration of a LDAP backend TLS clause:

```
tls {
    # LDAP client's certificate
    certificate "ldap.client.cert";

    # LDAP client's private key
    certificate_key "ldap.client.key";

    # LDAP client's certificate's root CA
    client_ca_certificate "ldap.client.ca";

    # LDAP server's certificate's root CA
    server_ca_certificate "ldap.server.ca";

    # Custom certificate verification rules are not supported for LDAP server
}
```

RADIUS backend

This backend acts as a RADIUS proxy, forwarding authentication requests to other RADIUS servers.

Example configuration of a RADIUS proxy backend:

```
radius "RADIUS_PROXY_EXAMPLE_ORG" {
    # backend server selection
    server-selection round-robin;

    server "radius1.example.org" {
        # RADIUS shared secret
        secret "ExampleSecret";

        # Request timeout in milliseconds
        timeout 7000;

        # how many times to retry the request
        retries 0;

        # use status-server polling
        # true = on, false = off
        status false;

        # how many sockets/connections at maximum to open
        #connections 16;

        connect {
            # transport protocol: udp/tcp/tls
            protocol udp;

            # server's IP address
            ip 203.0.113.111;

            # alternatively server's hostname
            #hostname radius1.example.org;

            # destination UDP port
            port 1812;

            # Optional: receive and send buffer size in bytes
            #buffer 1048576;
        } # connect
    } # server

    server "radius2.example.org" {
        secret "ThisIsAnExampleSecret";
        timeout 7000;
        retries 0;
        status false;
        connect {
            protocol udp;
            ip 203.0.113.112;
            #hostname radius2.example.org;
            port 1812;
            buffer 1048576;
```

```
        } # connect
    } # server

    # Modify/filter RADIUS request before proxying
    pre-proxying {
        # filter the following attributes before proxying
        filter {
            #cisco-avpair;
            Tunnel-Type;
            Tunnel-Medium-Type;
            Tunnel-Private-Group-ID;
        }
        # modify the following attributes before proxying
        modify {
            radiusproxy.request.attr.Operator-Name := "4EXAMPLE_COM:FI";
        }
    } # pre-proxying

    # Modify/filter RADIUS reply
    post-proxying {
        filter {
            # filter all vendor specific attributes
            #vendor-specific;
            #cisco-avpair;
            # filter attributes for VLAN assignment
            Tunnel-Type;
            Tunnel-Medium-Type;
            Tunnel-Private-Group-ID;
        } # filter
    } # post-proxying
} # radius "RADIUS_PROXY_EXAMPLE_ORG"
```

Server selection

This attribute determines how the server selects a backend server. Possible values are:

- `round-robin`: Servers are selected in a round-robin fashion.
- `fallback`: Servers are selected in order, with fallback to the next server if the current server is unavailable.
- `no-fallback`: Only the first server is used. If it is unavailable, the request fails.

Server selection is defined as a string, such as:

```
radius "EXAMPLE-SERVER" {
    server-selection round-robin;

    # other server configuration options...
}
```

Server

This is where the proxying server is defined. The server clause has a name parameter which is string.

Timeout

The timeout parameter specifies the amount of time, in milliseconds, that the RADIUS proxy will wait for a response from the upstream RADIUS server before considering the request to have timed out. This parameter is defined as a u32, which is an 32-bit unsigned integer.

Retries

The `retries` parameter is designed to provide a degree of fault tolerance in case of temporary network issues or upstream server unavailability. If the proxy doesn't receive a response from the upstream server within a certain timeout period, it will retry the request. This parameter is defined as a u8 which is an 8-bit unsigned integer.

Status

This parameter enables or disables status-server polling for the next hop RADIUS backend server. When set to true, Radiator will periodically poll the next hop server to check its status. When set to false, status-server polling is disabled. This is useful for monitoring the availability and health of the next hop RADIUS backend server. The status parameter is defined as a boolean.

Connections

This optional parameter specifies the number of concurrent connections to maintain with the RADIUS backend server. This allows the backend server to handle multiple authentication requests simultaneously, improving performance and throughput. The connections parameter is defined as a u16, meaning it's an unsigned 16-bit integer.

Idle timeout

When protocol is set to TCP or TLS, this optional parameter specifies the maximum amount of time, in seconds, that a connection to the RADIUS backend server can remain idle before it is closed. This helps to conserve resources and prevent connections from being held open indefinitely. The idle_timeout parameter is defined as a u32, meaning it's an unsigned 32-bit integer. If not specified, connections will remain open indefinitely, or until the server closes them.

Connect

This clause defines the connection parameters for the RADIUS backend server, specifying how Radiator establishes a connection to a server that is specified with specific parameters. These parameters include the transport protocol, either the IP address or the hostname, and the port number. This clause is not mandatory. It is used when RADIUS messages need to be forwarded to another RADIUS server. This is common in roaming scenarios, where authentication requests from local users are handled locally, while requests from roaming partners are forwarded to a remote RADIUS server.

Example configuration of `connect` clause:

```
connect {
    # transport protocol: udp/tcp/tls
    protocol udp;

    # server's IP address
    ip 203.0.113.111;
```

```
    # alternatively server's hostname
    #hostname radius1.example.org;

    # destination UDP port
    port 1812;

    # Optional: receive and send buffer size in bytes
    #buffer 1048576;
} # connect
```

Policy

This is the start of a policy clause. The clause has a name which is defined as a string. A single policy can contain multiple handlers from which a first handler to match will be selected to handle the received request.

Example configuration of a policy clause:

```
policy "Local network access" {
    handler "WLAN Controllers" {
        # (Optional) Only requests matching the conditions will be handled by this handler
        conditions all {
            radius.client == "WLAN Controllers";
        }

        # Handler configuration
        # ...
    }

    handler "Network switches" {
        # (Optional) Only requests matching the conditions will be handled by this handler
        conditions all {
            radius.client == "Network switches";
        }

        # Handler configuration
        # ...
    }

    # A handler without conditions will handle any request
    handler "Default handler" {
        # Handler configuration
        # ...
    }
}
```

Policies are configured within an aaa clause which can include multiple policies:

```
aaa {
    policy "example1" {

    }
```

```
    policy "example2" {

    }
}
```

Policy conditions

Each policy can be configured with an optional conditions clause. This clause is used to constrain which requests will be handled with it. If conditions are not defined, the policy will handle any request. A condition rule consists of a namespace attribute, comparison operator and one or more values to compare.

Policy and handler conditions are defined as:

```
conditions <all | any | none> {
    <namespace attribute> <comparison operator> <value>;
    <namespace attribute> <comparison operator> [<value> <value> <value>];
    ...
}
```

Supported matching strategies are:

- `all`: All condition rules must match
- `any`: Any condition rule must match
- `none`: None of conditions rules must match The format for configuring conditions for handlers is identical to configuring conditions for policies.

Handler

This is the start of a handler clause. The handler specifies how incoming requests are processed. The handler can, for example, be used to specify authentication methods, authorization rules, post-AA actions and logging. Multiple handlers can be defined within a policy.

A handler configuration consists of one or more blocks returning a value which is either none, accept, or reject.

By default, directives and statement clauses within a clause are executed in order until a reject is returned or the return value of the last directive or statement clause is returned. Changing this can be done by specifying an inner statement clause with a different strategy.

Example configuration of a handler clause inside a default policy:

```
policy "default" {

    # Policy handler named "default"
    handler "default" {
        # Try to authenticate requests
        authentication {
            # Search for username from backend "users"
            backend "users";

            # Try to authenticate user with PAP
```

```
            pap;
        }

        # Authorize authenticated requests
        authorization {
            # Set a reply message based on user's role
            map user.role {
                "admin" => {
                    message "Welcome admin!";
                }
                "guest" => {
                    message "Welcome guest!";
                }
            }

            # Explicitly accept, if user's roles didn't match
            accept;
        }

        # Log authentication requests
        post-authentication {
            # Log with AAA logger "auth"
            log "auth" {
                format "%{datetime.timestamp} method=%{aaa.method} username=\"%{aaa.identity}\"
            }
        }

        # Explicitly just accept any accounting requests
        accounting {
            accept;
        }
    }
}
```

Handler conditions

Each handler can be configured with an optional conditions clause. This clause is used to constraint which requests will be handled with it. If conditions are not defined, the handler will handle any request. A condition rule consists of a namespace attribute, comparison operator and one or more values to compare.

For supported matching strategies, see Policy conditions.

Example configuration of a handler conditions clause:

```
conditions all {
    aaa.accounting == false;
}
```

Authentication

An authentication clause defines the stages and configuration options for processing authentication requests within a handler. There are three different authentication clauses that can be configured:

- `pre-authentication`: (Optional) This stage is executed before the main authentication process.

- `authentication`: This stage handles the actual authentication. This stage is mandatory for processing authentication requests.

```
authentication {
    # search for username from backend "USERS_INTERNAL_FILE"
    backend "USERS_INTERNAL_FILE";

    # try to authenticate user with PAP
    pap;
}
```

- `post-authentication`: (Optional) This stage is executed after both authentication and authorization processes are complete.

```
post-authentication {
    template "LOG_AUTHENTICATION";
}
```

Authorization

The optional Authorization stage defines rules and actions to determine what resources a user can access after successful authentication. This stage is executed only if the authentication process returns an "accept" response.

Example configuration of an authorization clause:

```
authorization {
    # add RADIUS attributes to RADIUS reply
    modify {
        radius.reply.attr.Session-Timeout += 86400;
        radius.reply.attr.Acct-Interim-Interval += 600;
        radius.reply.attr.WBA-Identity-Provider += "EXAMPLE_COM:FI";
    }
}
```

Accounting

The Accounting section defines the stages for processing accounting requests in a handler.

`pre-accounting`: (Optional) This stage is executed before the main accounting process.

`accounting`: This stage handles the main accounting process. Example configuration of an accounting clause:

```
accounting {
    # acknowledge all accounting even if this
    # handler should not receive any
    accept;
}
```

`post-accounting`: (Optional) This stage is executed after the main accounting process is complete. Example configuration of a post-accounting clause:

```
post-accounting {
    template "LOG_ACCOUNTING";
}
```

Handler statements

Following statement clauses are supported:

- `first`: Return on first accept or reject.
- `any`: Return on first accept.
- `while`: Return on first reject (this is a default for all blocks).
- `all`: Returns reject on first none or reject.
- `none`: Returns reject on first accept.
- `each`: Return with the last return value.

Examples handler statement configurations:

```
# Authenticate with the first method used
first {
    pap;
    chap;
    mschapv2;
}
```

```
# Try to find a user from different backends
any {
    backend "ldap1";
    backend "ldap2";
}
```

Handler directives

Handler directives define specific actions that the RADIUS server should take when processing a request.

Following directives are supported:

Request handling directives These directives determine how the server should respond to the request:

- `accept`: The policy explicitly allows the request. When this directive is used, the server unconditionally accepts the request and typically sends an “Access-Accept” response back to the client.
- `reject`: This directive unconditionally denies the request. The server responds with an “Access-Reject” message.
- `challenge`: Instead of simply accepting or rejecting, the server responds with a challenge. This is used in multi-factor authentication scenarios where the client must provide additional credentials or perform extra steps before access is granted.

- `message` : Message is used to add additional information to the response.
- `reason` : Reason is also used to add additional information to the response that provides additional context. For example, the reason might explain why a request was rejected.

Backend authentication and forwarding These directives control authentication methods and backend queries:

- `backend` : Directs the policy handler to forward the request to an external backend for further processing or to look up additional attributes.
- `pap` : Try to authenticate the request with PAP (plaintext password).
- `chap` : Try to authenticate the request with CHAP.
- `mschap` : Try to authenticate the request with MSCHAP.
- `mschapv2` : Try to authenticate the request with MSCHAPv2.

Logging These directives allow logging for debugging or auditing:

- `log` : Logs the transaction with an AAA logger.

Context attribute manipulation These directives allow modifying context attributes:

- `modify` : Modifies context attributes.
- `set` : Assigns a value to an attribute.
- `replace` : Replaces an existing attribute with a new value.
- `rewrite` : Modifies the value of an attribute based on its current value.
- `append` : Appends a new value into an attribute.

> Copy and filter can only be used in backend RADIUS pre-proxying and post-proxying handlers.

- `copy` : Copies wanted attributes into to be proxied RADIUS request or from a received RADIUS proxy response.
- `filter` : Filters unwanted attributes from a to be proxied RADIUS request or from a received RADIUS proxy response.

Logging

The logging configuration includes two clauses which are application and aaa. Here is an example configuration of the logging clause:

```
logging {
    # Application logging
    application {...}

    # AAA logger(s)
    aaa {...}
}
```

Logging sources

This section introduces the two different logging sources which are AAA and Application. Logging sources are configured within a logging clause.

AAA logging

This clause is used for configuring one or multiple AAA loggers. The AAA loggers are responsible for recording and monitoring authentication, authorization, and accounting activities within the network.

AAA loggers are defined with separate logger clauses inside the AAA block. The logger clause defines a named AAA logger that can be configured to log messages using output methods such as file, memory, and console. Each logger is configured with a name which is defined as a string. A syslog clause can also be defined inside the logger. For more information, see Syslog.

Example configuration of a logger named "auth":

```
logger "auth" {
    # File logger
    file {...}

    # In-memory logger
    memory {...}

    # Local syslog logger
    syslog {...}
}
```

Application logging

This defines the application logger. The application logger is used to record the events in the applications lifecycle.

Example configuration of application logging:

```
application {
        # File logger
        file {...}

        # In-memory logger
        memory {...}

        # Console logger
        console {...}

        # Local syslog logger
        syslog {...}
    }
```

Logging destinations

This section introduces the different logging destinations which are file, console, and memory.

File

This is used to configure the file logger which stores log messages in a file you specify. The filename is defined as a string.

Example configuration of a file logger clause:

```
file {
    filename "filelog.log";
}
```

Console

This is used to configure the console logger which outputs log messages into console.

Example configuration of console logger:

```
console {
    loglevel warning;
}
```

Note that by default, the loglevel inside the console clause is set to info. Therefore, the console logger can also be configured without explicitly specifying a loglevel:

```
console;
```

Memory

This is used to configure the memory logger which temporarily stores log messages in memory.

Example configuration of a memory logger clause:

```
memory {
    loglevel warning;
    size 32768;
}
```

Note that by default, the loglevel inside the memory clause is set to info, and the size is set to 16384. Therefore, the memory logger can also be configured without explicitly specifying a loglevel and size:

```
memory;
```

Common logging attributes

This section introduces common logging attributes.

Size

The size attribute is used in in-memory logging. This specifies the maximum number of log entries to be stored in memory. The size attribute is defined as a u32, which is a 32-bit unsigned integer.

Loglevel

This specifies the priority level that depends on the severity of the message. The loglevel attribute is used in all logging methods including syslog. There are six possible values for the loglevel:

- `off` : Disables logging.
- `error` : Logs only error messages, which indicate serious problems.
- `warning` : Logs warning and error messages, which indicate potential issues that may need attention.
- `info` : Logs informational, warning, and error messages, providing a general overview of the system's status.
- `debug` : Logs detailed debugging information, including informational, warning, and error messages.
- `trace` : Logs very detailed tracing information, including all debug, informational, warning, and error messages, for in-depth troubleshooting.

Syslog

This optional clause creates a syslog logger, which logs all messages with a specified priority level or higher to the syslog system.

Messages are logged to syslog with severity levels that depend on the severity of the message. There are 8 defined priority levels in syslog, and they are logged to the equivalent syslog priority. The priority levels with their corresponding values and descriptions are:

- Emergency (0): System is unusable
- Alert (1): Immediate action needed
- Critical (2): Critical conditions
- Error (3): Error conditions
- Warning (4): Warning conditions
- Notice (5): Normal but significant conditions
- Informational (6): Informational messages
- Debug (7): Debug-level messages

Example configuration of a syslog logger:

```
syslog {
    loglevel info;
    facility daemon;

    # optional syslog unix domain socket path
    #filename /var/run/syslog;
}
```

Facility

This specifies the name of the system that is logging the syslog message. The facility attribute is defined as a string. The way that a message is handled may differ depending on the facility. Each of these facilities help in organizing and routing syslog messages based on their source.

Facility keywords in syslog are defined as strings. Each facility has a corresponding numerical code. The possible values with their corresponding facility codes are:

- `kern` (0): Kernel messages
- `user` (1): User-level messages
- `mail` (2): Mail system
- `daemon` (3): System daemons
- `auth` (4): Security/authentication messages
- `syslog` (5): Messages generated internally by syslog
- `lpr` (6): Line printer subsystem
- `news` (7): Network news subsystem
- `uucp` (8): UUCP subsystem
- `cron` (9): Cron subsystem
- `authpriv` (10): Security/authentication messages
- `ftp` (11): FTP daemon
- `ntp` (12): NTP subsystem
- `security` (13): Log audit
- `console` (14): Log alert
- `solaris-cron` (15): Scheduling daemon
- `local0` (16): Locally used facility 0
- `local1` (17): Locally used facility 1
- `local2` (18): Locally used facility 2
- `local3` (19): Locally used facility 3
- `local4` (20): Locally used facility 4
- `local5` (21): Locally used facility 5
- `local6` (22): Locally used facility 6
- `local7` (23): Locally used facility 7

Note that the mapping between facility codes and keywords is not uniform across different operating systems and syslog implementations. For additional information, refer to your operating system documentation.

Extensible Authentication Protocol (EAP)

Extensible Authentication Protocol (EAP) is a standard for defining and extending authentication protocols. EAP is defined by RFC 3748, and RFC 2869 defines how EAP authentication messages are carried in RADIUS packets. Radiator complies with these standards, and can be extended to handle any EAP-compliant protocol. Because EAP is designed to be easily extensible, a number of EAP protocols have been defined for various special requirements This includes mutual authentication between client and RADIUS

server, and encryption of the authentication conversation. EAP over RADIUS is commonly used as the authentication protocol for 802.1X wired and wireless networks. For more information, see IETF website.

Conventional RADIUS requests send the User-Name in one RADIUS attribute and the User-Password in another attribute. All EAP-over-RADIUS requests send their authentication information in the EAP-Message attribute. Usually the client sends some EAP protocol information in the EAP-Message attribute in a RADIUS Access-Request message, and the RADIUS server asks for some more information from the client by sending back another EAP-Message in a RADIUS Access-Challenge. When the server has enough information from the client, the RADIUS server replies with an Access-Accept or an Access-Reject message. Some EAP protocols, such as TLS, TTLS and PEAP, often require a number of messages (10 or more) to be exchanged between client and RADIUS server during authentication. Such a group of EAP-over-RADIUS messages while authenticating a user is called a conversation.

Note that many EAP protocols hide or encrypt the identity of the real user name of the user being authenticated, and send a generic name (typically anonymous) as the visible User-Name in the RADIUS requests. Radiator supports five EAP protocols which are EAP-MD5-CHALLENGE, PEAP, TEAP, EAP-TLS, and EAP-TTLS. This section describes their characteristics and how Radiator can be configured to support them.

In order to use EAP protocols that utilize TLS, Radiator requires you to configure a unique Server Certificate. For more information about Public and Private certificates and how to obtain them, see EAP-TLS.

EAP configuration

This section introduces the parameters used for EAP configuration as well as basic configuration instructions.

Each EAP protocol needs to be defined inside an eap clause which is defined in this example configuration:

```
aaa {
    # AAA policy named "default"
    policy "default" {
        # Policy handler named "default"
        handler "default" {
            # Try to authenticate requests
            authentication {
                # EAP protocols are defined in this eap clause
                eap {...}
            }
        }
    }
}
```

EAP-MD5-Challenge

EAP-MD5-challenge (sometimes called just EAP-MD5) provides a simple challenge-response mechanism for authenticating a client. EAP-MD5-Challenge is compatible with CHAP authentication. EAP-MD5-Challenge authentication happens following these basic steps:

1. The RADIUS server sends a random challenge to the client.

2. The client forms an MD5 hash of the user's password and the challenge and sends the result back to the server.
3. The server then validates the MD5 hash using the known correct plaintext password from the user database.
4. If the client's hash matches the server's hash, the authentication is successful.

Example configuration of EAP-MD5-Challenge with CHAP authentication:

```
eap-md5 {
    authentication {
        # Search for EAP identity from backend "users"
        backend "users-file";

        # Try to authenticate user with CHAP
        chap;
    }
}
```

PEAP

Like EAP-TLS, PEAP (sometimes called EAP-PEAP) uses Public Key Infrastructure (PKI) digital certificates. Unlike TLS, it only uses a Server Certificate so the client can validate the server, and then establish a secure, encrypted communications channel with the RADIUS server. When this channel is established, it is used to tunnel encrypted EAP messages to the RADIUS server. So PEAP authentication happens in 2 phases following these basic steps:

1. The PEAP client and RADIUS server establish a communications channel via the RADIUS protocol.
2. The RADIUS server sends its Server PKI Certificate to the client.
3. The client verifies that the server certificate is valid and is the correct certificate for the RADIUS server it is communicating with. It uses the Root Certificate of the Certificate Authority that issued the Server Certificate to validate the Server Certificate. (Root Certificates for most Public Certificate Authorities are built in to most clients. If the Server Certificate was issued by a Private Certificate Authority, the client requires a copy of the Root Certificate to be installed in order to validate the Server Certificate.)
4. If the client validates the server certificate, it then sends one or more EAP requests through the encrypted TLS tunnel. The type of inner EAP request depends on the PEAP client configuration, but the most common types of inner EAP requests are EAP-MSCHAP-V2 and EAP-TLS.
5. Radiator handles the inner EAP-TTLS request in the eap-ttls authentication and authorization handlers.
6. The result of the inner authentication is sent back to the client through the TLS tunnel.

You can also configure Radiator to convert an inner EAP-MSCHAP-V2 request into a conventional RADIUS-MSCHAP-V2 request, which means that Radiator can serve as a gateway between PEAP clients and a non-EAP enabled RADIUS server.

Example configuration of PEAP authentication:

```
eap-peap {
    tls {
        # AAA server's certificate
```

```
        certificate "eap.server.cert";

        # AAA server's private key
        certificate_key "eap.server.key";

        # AAA server's certificate's root CA
        server_ca_certificate "eap.server.ca";

        # Require client certificates
        #require_client_certificate true;

        # Client certificates' root CA
        client_ca_certificate "eap.client.ca";
    }

    # Try to authenticate EAP-PEAP inner requests
    authentication {
        eap {
            # Try to authenticate EAP-PEAP inner requests with EAP-GTC
            eap-gtc {
                authentication {
                    # Search for EAP identity from backend "users-file"
                    backend "users-file";

                    # EAP-GTC is compatible with PAP authentication
                    # Try to authenticate user with PAP
                    pap;
                }

                authorization {
                    # Copy user roles into variables which are available everywhere
                    set vars.role = user.role;
                }
            }
        }
    }

    post-authentication {
        # Log inner authentication requests
        post-authentication {
            # Log with AAA logger "auth"
            log "auth" {
                format {
                    "%{datetime.timestamp} "
                    "method=%{aaa.method} "
                    "username=\"%{aaa.identity}\" "
                    "identity=\"%{eap.identity}\" "
                    "eap=%{eap.method} "
                    "result=%{aaa.result} "
                    "reason=\"%{aaa.reason}\""
                }
            }
        }
    }
}
```

TEAP

TEAP (sometimes called EAP-TEAP) is an extension of EAP designed to provide secure authentication over wireless networks. It operates similarly to EAP-TTLS and PEAP but includes additional features tailored for enhanced security and flexibility in wireless environments.

So TEAP authentication happens following these basic steps:

1. The TEAP client and RADIUS server establish a communications channel via the RADIUS protocol.
2. The RADIUS server sends its Server PKI Certificate to the client.
3. The client verifies that the server certificate is valid and is the correct certificate for the RADIUS server it is communicating with. It uses the Root Certificate of the Certificate Authority that issued the Server Certificate to validate the Server Certificate. (Root Certificates for most Public Certificate Authorities are built in to most clients. If the Server Certificate was issued by a Private Certificate Authority, the client requires a copy of the Root Certificate to be installed in order to validate the Server Certificate.)
4. Upon successful validation of the server certificate (and optionally the client certificate), a secure TLS tunnel is established between the client and the RADIUS server.
5. Radiator handles the inner EAP-TTLS request in the eap-ttls authentication and authorization handlers.
6. The result of the inner authentication is sent back to the client through the TLS tunnel.

Example configuration of TEAP authentication:

```
eap-teap {
    tls {
        # AAA server's certificate
        certificate "eap.server.cert";

        # AAA server's private key
        certificate_key "eap.server.key";

        # AAA server's certificate's root CA
        server_ca_certificate "eap.server.ca";

        # (Don't) Require client certificates
        require_client_certificate true;

        # Client certificates' root CA
        client_ca_certificate "eap.client.ca";

        keylog_filename "/var/lib/radiator/keylog/radiator-server.keylog";
    }

    # Try to authenticate EAP-TEAP inner requests
    authentication {
        # Explicitly accept
        accept;
    }

    authorization {
        # Explicitly accept
        accept;
    }
```

```
    # Log inner authentication requests
    post-authentication {
        # Log with AAA logger "auth"
        log "auth" {
            format {
                "%{datetime.timestamp} "
                "method=%{aaa.method} "
                "username=\"%{aaa.identity}\" "
                "identity=\"%{eap.identity}\" "
                "eap=%{eap.method} "
                "result=%{aaa.result} "
                "reason=\"%{aaa.reason}\""
            }
        }
    }
}
```

EAP-TLS

EAP-TLS uses Public Key Infrastructure (PKI) digital certificates to provide mutual authentication between the EAP client and the RADIUS server. A PKI certificate is a file created by a program called a Certificate Authority. The certificate contains the name of the server or user that has been issued to. The EAP client and RADIUS server use the certificates to verify that the other party is indeed who it claims to be. In EAP-TLS, a PKI certificate is required for the Radiator RADIUS server and for each and every EAP-TLS client.

You can obtain certificates from a Public Certificate authority such as DigiCert. The advantage of Public Certificates is that they will generally be recognized by any client or server without taking any special steps. A disadvantage of Public certificates is that you usually have to pay an annual fee for each one. With a Private Certificate Authority, you can generate your own server and client certificates for free, but you will generally have to install the ‘Root Certificate’ from your Certificate Authority on each client before it will recognize a private server certificate. Private Certificates are considered by many to be more secure that Public Certificates.

The basic steps of EAP-TLS authentication are:

1. The EAP-TLS client and RADIUS server establish a communications channel via the RADIUS protocol.
2. The RADIUS server sends its Server PKI Certificate to the client.
3. The client verifies that the server certificate is valid and is the correct certificate for the RADIUS server it is communicating with. It uses the Root Certificate of the Certificate Authority that issued the Server Certificate to validate the Server Certificate. (Root Certificates for most Public Certificate Authorities are built in to most clients. If the Server Certificate was issued by a Private Certificate Authority, the client requires a copy of the Root Certificate to be installed in order to validate the Server Certificate.)
4. If the client validates the server certificate, it then sends the user's PKI certificate to the RADIUS server.
5. The RADIUS server verifies that the client certificate is valid and is the correct certificate for the user name that is being authenticated. The RADIUS server can be configured to validate Private Client Certificates using a locally installed copy of the Root Certificate of the Certificate Authority that issued the client certificate.
6. If the RADIUS server validates the client certificate then the authentication is successful, and the client is permitted to be connected to the network.

Example configuration of EAP-TLS authentication:

```
eap-tls {
    tls {
        # AAA server's certificate
        certificate "eap.server.cert";

        # AAA server's private key
        certificate_key "eap.server.key";

        # AAA server's certificate's root CA
        server_ca_certificate "eap.server.ca";

        # Require client certificates
        #require_client_certificate true;

        # Client certificates' root CA
        client_ca_certificate "eap.client.ca";

        # (Optional) Client certificate's verification policy
        verification {
            # Lookup user based on client certificate's CN
            backend {
                # Use backend named "users-file"
                name "users-file";

                # Do backend query based cert subject email address
                query cert.subject.emailaddress;
            }
            # Reject the certificate if any of following conditions fails
            if any {
                # Basic certification verification failed
                cert.valid != true;
                # User not found
                user.username == none;
            } then {
                reject;
            } else {
                accept;
            }
        }
    }
}
```

EAP-TTLS

Like EAP-TLS, EAP-TTLS uses Public Key Infrastructure (PKI) digital certificates. Unlike TLS, it only uses a Server Certificate so the client can validate the server, and then establish a secure, encrypted communications channel with the RADIUS server. When this channel is established, it is used to tunnel conventional RADIUS attributes, such as User-Name, User-Password etc. to the RADIUS server. EAP-TTLS authentication happens in 2 phases following these basic steps:

1. The EAP-TTLS client and RADIUS server establish a communications channel via the RADIUS protocol.
2. The RADIUS server sends its Server PKI Certificate to the client.

3. The client verifies that the server certificate is valid and is the correct certificate for the RADIUS server it is communicating with. It uses the Root Certificate of the Certificate Authority that issued the Server Certificate to validate the Server Certificate. (Root Certificates for most Public Certificate Authorities are built in to most clients. If the Server Certificate was issued by a Private Certificate Authority, the client requires a copy of the Root Certificate to be installed in order to validate the Server Certificate.)
4. If the client validates the server certificate, it then sends the real user name and password in a RADIUS request through the encrypted TLS tunnel. Any conventional RADIUS authentication system may be used depending on the client configuration, such as PAP, CHAP, MSCHAP, MSCHAP-V2 etc.
5. Radiator handles the inner EAP-TTLS request in the eap-ttls authentication and authorization handlers.
6. The result of the inner authentication is sent back to the client through the TLS tunnel.

Example configuration of EAP-TTLS authentication:

```
eap-ttls {
    tls {
        # AAA server's certificate
        certificate "eap.server.cert";

        # AAA server's private key
        certificate_key "eap.server.key";

        # AAA server's certificate's root CA
        server_ca_certificate "eap.server.ca";
    }

    # Try to authenticate EAP-TTLS inner requests
    authentication {
        # Search for username from backend "users-file"
        backend "users-file";

        # Try to authenticate user with any of following methods
        any {
            pap;
            chap;
            mschapv2;
            eap {
                eap-mschapv2;
            }
        }
    }

    # Log inner authentication requests
    post-authentication {
        # Log with AAA logger "auth"
        log "auth" {
            format {
                "%{datetime.timestamp} "
                "method=%{aaa.method} "
                "username=\"%{aaa.identity}\" "
                "identity=\"%{eap.identity}\" "
                "eap=%{eap.method} "
                "result=%{aaa.result} "
                "reason=\"%{aaa.reason}\""
            }
```

```
        }
    }
}
```

EAP-MSCHAP-V2

EAP-MSCHAP-V2 is an EAP version of the common MSCHAP-V2 authentication mechanism. It provides mutual authentication between client and server. It is most commonly used as the inner authentication protocol with EAP PEAP on Microsoft Windows clients.

EAP-MSCHAP-V2 can be used with any Radiator backend that has access to plaintext passwords, such as FILE, SQL, LDAP2 etc. It can also be used with LSA and NTML backends to authenticate with a Windows Local Security Authority, Windows Domain Controller etc.

License

The license clause is used to read the license configuration parameters. There is no default.

License configuration clause:

```
license {
    # use a license directory to look for licenses
    directory "/var/lib/radiator/licenses"
}
```

Dictionary

The dictionary file defines easy-to-read names for the attributes and values used in RADIUS messages. It defines how RADIUS attribute numbers map to readable attribute names, and how RADIUS value numbers map to readable value names. The dictionary also defines the type of data that each attribute can hold.

The dictionary file is an ASCII text file. Each definition occupies one line. A hash mark # marks the beginning of a comment. Comment and blank lines are ignored.

Example configuration of a dictionary clause:

```
dictionary {
    radius {
        filename "path/to/dictionary";
    }
}
```

RADIUS

This attribute defines the RADIUS dictionary file.

Filename

The name of the dictionary file. The filename attribute can also include the specified directory, e.g., /opt/radiator/server/dictionary. Filename is defined as a string.

Capture

Captures are used for detailed logging of RADIUS server packets. The capture allows dumping received and sent RADIUS messages to an output specified by the specific capture configuration. Possible output destinations include the console (stdout), memory, or a file (PcapNG format).

Example configuration of a capture that logs the messages into the console, memory, and a file:

```
captures {
    capture "test_capture" {
        console;

        memory;

        file {
            directory ".";
            filename "test_capture.pcapng";
        }
    }
}
```

A capture can be enabled, for example, within:

```
servers {
    radius {
        capture "test_capture";

        ...
    }
}
```

Capture destinations

This section introduces the three different capture output locations which are console, file, and memory.

Console

Outputs the captured RADIUS messages to the console (stdout).

Memory

Stores the captured RADIUS messages in memory.

File

Stores the captured RADIUS messages in a file. The file type is PcapNG.

Directory

Specifies the directory where the capture file will be stored. Parameter type is string.

Filename

Specifies the name of the file where captured data will be written. Parameter type is string.

Store

A store is an internal NoSQL key-value database built into the application. The directory location can be configured using the optional filename attribute, which is defined as a string. If the filename is not configured, the store will be saved to the working directory of the application.

Example configuration of a store clause:

```
store "example_store" {
    # Optional file path for the store
    filename "/some/path";
}
```

Stores can be configured within:

```
stores {
    store "example_store";
}
```

Supported namespaces, attributes, and data types

This section covers the supported namespaces, attributes, and data types.

Data types

Following data types are supported:

- `none` : No value: none
- `any` : Any value: any
- `boolean` : Boolean value: true or false
- `unsigned` : Unsigned number. Examples: 10, 6000
- `enum` : Unsigned numeric enum. Examples: START, radius, 1, 2
- `signed` : Signed number. Examples: -10, -6000, 10, 6000
- `float` : Floating point number. Examples: 3.5, 20.9
- `timestamp` : Timestamp. Examples: now, RFC 2822 format, RFC 3339 format
- `string` : Text string. Examples: "some text", word
- `bytes` : Bytes. Examples: 0xaa00ffee
- `ip` : IPv4 or IPv6 address. Examples: 10.10.10.10, 2001:db8:3333:4444:5555:6666:7777:8888
- `ip-prefix` : IPv4 or IPv6 address prefix. Examples: 10.10.10.0/24, 2001:db8::0/32

- `regex` : Regular expression. Example: /^example.(com|org)$/

If a namespace attribute has no value, none is returned.

Format string and filters

Enclosing a namespace attribute within "%{ }" evaluates its value within a string. In addition, optional filter(s) can be used to modify the evaluated value. Examples:

```
"username=%{ aaa.identity }"       Gets a request username
"%{ auth.response }"               Get an authentication response
```

Following optional filters are supported:

- `hex` : Converts a string or a byte value into a hexadecimal string
- `string` : Converts a non-string or a byte value into a string
- `uppercase` : Converts a string value into uppercase
- `lowercase` : Converts a string value into lowercase
-

Example:

```
"%{ util.rand.20 | hex}"           Gets 20 random bytes as a hex string
```

Namespaces

Below, '*' denotes any namespace, attribute name, or data type. **Namespace "id":**

```
id        number        read        Request identifier
```

Namespace "datetime":

```
datetime.timestamp        timestamp        read        Current time
```

Namespace "aaa":

```
aaa.protocol            enum        read        A protocol of the request
aaa.trace               boolean     read        Is tracing enabled for the request?
aaa.accounting          boolean     read        Is the request an accounting request?
aaa.policy              string      read        The name of AAA policy handling the request
aaa.handler             string      read        The name of AAA policy handler handling the rec
aaa.identity            string      read/write  A current username/identity of the request
aaa.identity.name       string      read        A name part of a current username/identity
aaa.identity.realm      string      read/write  A realm part of a current username/identity
aaa.identity.realm.tld  string      read        TLD part of the realm
aaa.method              enum        read        Authentication protocol/method
aaa.message             string      read/write  Reply message
```

```
aaa.result              enum        read        AAA result
aaa.reason              string      read        An error or a reason
```

Namespace "auth":

```
auth.protocol       enum        read/write  Authentication protocol
auth.challenge      string      read/write  Authentication challenge
auth.response       bytes       read/write  Authentication response
auth.result         string      read/write  Authentication result
```

Namespace "acct":

```
acct.status         enum        read        Accounting request status
acct.timestamp      timestamp   read        Accounting request timestamp
acct.duration       number      read        Accounting session duration in seconds
acct.input.packets  number      read        Accounting session packets in
acct.input.bytes    number      read        Accounting session bytes in
acct.output.packets number      read        Accounting session packets out
acct.output.bytes   number      read        Accounting session bytes out
```

Namespace "user":

```
user.username         string              read/write        User entry's name
user.password         string              read/write        User's password
user.group            string array        read/write        User's groups
user.role             string array        read/write        User's roles
user.backend          string              read              Backend's name from which user was
```

Namespace "vars":

```
vars.*      *       read/write  Custom variables
```

Namespace "util":

```
util.rand.X        bytes       read        Reads X bytes of random
```

Namespace "tls":

```
tls.version        enum        read        Negotiated TLS protocol version
tls.cipher         enum        read        Negotiated TLS cipher
```

Namespace "cert":

```
cert.valid          boolean        read    Did basic certificate validation succeed?
cert.error          string         read    Verification error
cert.server         string         read    In TLS client context, server's name or IP address
cert.issuer         string         read    Certificate's issuer's name
```

```
cert.subject        string         read    Certificate's subject's name
cert.serial         number         read    Certificate's serial number
cert.issued         timestamp      read    Certificate's issue timestamp
cert.expires        timestamp      read    Certificate's expiry timestamp
cert.policy         string array   read    Certificate's Policy OIDs
cert.sha256         bytes          read    SHA256 hash of the certificate
cert.issuer.*       string array   read    Issuer name's individual components (dc, c, st, l, c
cert.subject.*      string array   read    Subject name's individual components (dc, c, st, l,
cert.subject_alt.*  string array   read    Subject's alternative names (email, dns, dn, uri, ip
cert.ca.*           *              read    Certificate's first CA certificate
cert.ca[N].*        *              read    Certificate's Nth CA certificate
```

CA certificate has the same attributes as the cert. **Namespace "radius":**

```
radius.client       string      read        Client's name from which the request was received
radius.client.ip    ip          read        IP address from which the request was received
radius.server       string      read        Server's name which received the request
radius.server.tls   boolean     read        Was request received over TLS?
radius.attr.*       *           read/write  Radius request/reply attributes
```

Attribute's data type is defined by the RADIUS dictionary used. Examples:

```
radius.attr.user-name
radius.attr.tunnel-type:1           Tunnel-Type attribute with a tag value 1
radius.attr.framed-route[0]         First Framed-Route attribute
radius.attr.framed-route[n]         Last Framed-Route attribute
radius.attr.framed-route[*]         All Framed-Route attributes
radius.attr.cisco.avpair            Cisco AV-pair attribute
```

Namespace "radiusproxy":

```
radiusproxy.server       string      read        Radius proxy server's name to which send the r
radiusproxy.server.tls   boolean     read        Is Radius proxy server using TLS connection?
radiusproxy.attr.*       *           read/write  Radius proxy reply/request attributes
```

Attribute's data type is defined by the RADIUS dictionary used. Examples:

```
radiusproxy.attr.user-name
radiusproxy.attr.operator-name
radiusproxy.attr.tunnel-type:1      Tunnel-Type attribute with a tag value 1
radiusproxy.attr.cisco.avpair[0]    First Cisco AV-pair attribute
radiusproxy.attr.cisco.avpair[n]    Last Cisco AV-pair attribute
radiusproxy.attr.cisco.avpair[*]    All Cisco AV-pair attributes
```

Namespace "eap":

```
eap.identity        string      read    EAP identity
eap.method          enum        read    EAP method
```

Namespace "eap-ttls":

```
eap-ttls.attr.*     *           read/write  EAP-TTLS request/response attributes
```

Attribute's data type is defined by the RADIUS dictionary used. Examples:

```
eap-ttls.attr.user-name
eap-ttls.attr.user-password
eap-ttls.attr.chap-password
eap-ttls.attr.eap-message
```

Namespace "eap-teap":

```
eap-teap.username       string  read    EAP-TEAP basic password authentication response usernam
eap-teap.identity-type  enum    read    EAP-TEAP Identity-Type TLV (user/machine)
```

Namespace "parent":

```
parent.*    *       read/write  Parent context namespace
```

Examples:

```
parent.aaa.identity
parent.radius.client
```

Namespace "root":

```
root.*      *       read/write  Root context namespace
```

Examples:

```
root.aaa.identity
root.radius.client
```